SWE-CLAR: CAN CODING AGENTS CLARIFY INCOMPLETE CLIENT REQUIREMENTS AND DELIVER GREENFIELD SOFTWARE?

Anonymous authors
Paper under double-blind review

ABSTRACT

Greenfield software development in freelance settings requires coding agents to turn incomplete client requests into working software. This makes requirement clarification essential, yet clients may be unable to provide continuous feedback over long development cycles, and receiving their answers does not guarantee correct understanding of complex requirements. These challenges motivate two benchmark design choices: clarification precedes code implementation, and its effectiveness is assessed through explicit tests of requirement understanding. Guided by these choices, we introduce SWE-Clar, a benchmark grounded in real, high-value client commissions to evaluate requirement clarification and code implementation in this setting. It comprises 100 tasks based on real software development jobs on Upwork, each with at least US$4,000 in total payments. Each task begins with multi-round interaction between a coding agent and a client agent, followed by separate runs for code implementation with acceptance testing and clarification assessment. For the latter, we construct 512 questions paired with reference answers from real client–freelancer interactions, covering both explicitly stated requirements and those inferred from the conversational context. Both runs begin from the same clarification context but remain isolated, preventing assessment questions from guiding code implementation. Experiments with nine frontier models show that even GPT-6 Astra, which has the highest observed implementation score, achieves a task pass rate of only 67%, underscoring the challenge of independent software delivery in this setting. The dataset and evaluation code are included in the supplementary material.

1 INTRODUCTION

As large language models (LLMs) improve at coding and reasoning, coding agents are expanding from generating simple code snippets (Chen et al., 2021) to tackling complex software engineering tasks (Yang et al., 2024). These advances create opportunities to automate software development services and deliver business value (Peng et al., 2023; Miserendino et al., 2025; Yi et al., 2025; Mazeika et al., 2025), particularly in greenfield software development, which involves building new software from scratch rather than modifying existing codebases (Ding et al., 2025; Shi et al., 2026). On freelance platforms such as Upwork[1], agents could act as freelancers to undertake high-value client commissions (Figure 1). However, these commissions typically begin with requests that leave critical functional and interface requirements unspecified (Nuseibeh & Easterbrook, 2000), so coding agents must proactively seek clarification through client interaction (Mu et al., 2024). Evaluating coding agents for greenfield software development in freelance settings therefore requires assessing both code implementation and requirement clarification.

Recent work has extended coding-agent evaluation beyond code implementation to include requirement clarification, enabling coding agents to interact with client agents to recover missing requirements (Wu & Fard, 2025; Fang et al., 2026; Vijayvargiya et al., 2026; Peng et al., 2026; Wu et al., 2026; Raghavendra et al., 2026; King & Flanigan, 2026). However, these benchmarks primarily focus on tasks derived from programming datasets and software repositories, leaving the demands of

[1] https://www.upwork.com/

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.

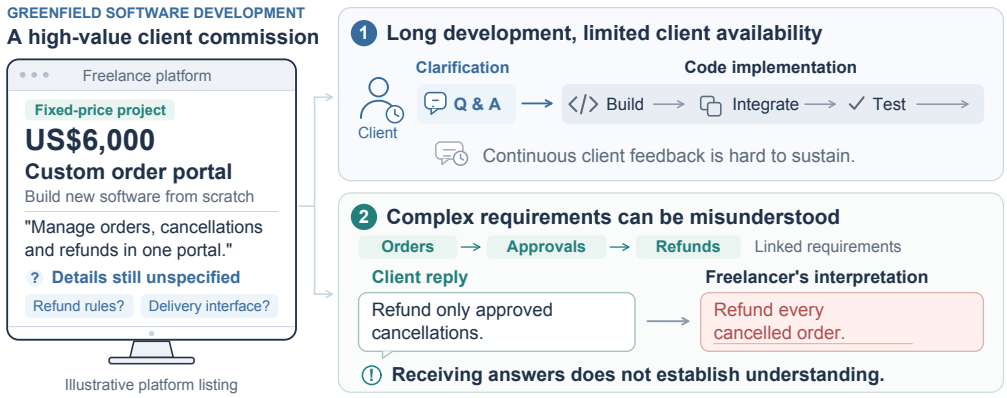

Figure 1: A fictional commission illustrates the motivation for SWE-Clar in greenfield software development in freelance settings: (1) long development makes continuous client feedback difficult to sustain; (2) complex requirements can be misunderstood even after receiving replies.

greenfield software development in freelance settings insufficiently represented. Our observations of real commissions for such greenfield software development projects highlight two characteristics, illustrated in Figure 1. First, building complete software projects from scratch often entails a prolonged development process, making continuous client feedback during code implementation difficult to sustain. Second, these projects involve multiple complex requirements that freelancers may not fully understand even after receiving answers to their questions. Consequently, benchmarks that assume continuous client feedback (Wu et al., 2026; Raghavendra et al., 2026) and assess clarification primarily through information acquisition (Fang et al., 2026; Vijayvargiya et al., 2026; Peng et al., 2026; Trinh et al., 2026) may not adequately assess coding agents' ability to fulfill these commissions independently.

These characteristics motivate two design principles for conducting and evaluating requirement clarification. First, clarification should precede code implementation: the coding agent should complete its interaction with the client agent before implementing the software without further guidance. Second, clarification should be assessed using multiple requirement questions paired with reference answers. The coding agent's responses should be scored against these reference answers to assess requirement understanding rather than information acquisition alone. This assessment should remain separate from code implementation to avoid providing additional implementation guidance.

Guided by these design choices, we introduce **SWE-Clar**, a benchmark of 100 greenfield software development tasks based on real software development jobs on Upwork, each with at least US$4,000 in total payments. For each task, a coding agent first engages in multi-round requirement clarification with a client agent. The resulting context then initializes two independent branches: one performs code implementation followed by acceptance testing, while the other evaluates clarification by scoring the coding agent's answers to requirement questions. To construct these questions, we use GPT-6 Astra to analyze recorded interactions between real clients and freelancers and identify requirements critical to successful software delivery, including those explicitly stated and those inferred from the broader conversational context. These requirements form the basis of 512 questions paired with reference answers, averaging approximately five per task. Together, these designs establish SWE-Clar as a benchmark grounded in real, high-value client commissions to evaluate requirement clarification and code implementation for greenfield software development in freelance settings. We evaluate nine frontier models on SWE-Clar to examine requirement clarification and independent code implementation in this setting. GPT-6 Astra has the highest observed implementation score, with a task pass rate of only 67%, underscoring SWE-Clar's value as a challenging testbed for identifying limitations in coding agents' ability to fulfill real client commissions.

- We distill two design principles for requirement clarification in greenfield software development in freelance settings: clarification before code implementation and independent assessment of requirement understanding through question answering.
- We introduce SWE-Clar, a benchmark grounded in real, high-value client commissions to evaluate requirement clarification and code implementation in this setting.

- We evaluate nine frontier models, demonstrating that independently completing these greenfield projects remains challenging even for the strongest evaluated model.

2 SWE-CLAR: BENCHMARK DESIGN

In this section, we introduce SWE-Clar, then describe requirement clarification and the two independent evaluation branches. We subsequently explain how the tasks and evaluation materials are constructed, and examine the dataset and its differences from representative benchmarks.

2.1 OVERVIEW

We aim to evaluate requirement clarification and code implementation for greenfield software development in freelance settings. A coding agent must both understand what a client wants and deliver software that meets those requirements. Following the design principles in Section 1, SWE-Clar therefore places client interaction before code implementation and assesses the agent's understanding separately from the software it produces.

Figure 2 shows the resulting workflow. Each task begins with an incomplete request for new software and a development workspace. A coding agent acts as the freelancer, while a separate client agent simulates the customer and answers from the complete requirements. The coding agent asks questions until it is ready to proceed. Its clarification context then initializes two independent runs: one implements the software and is scored through acceptance tests; the other answers requirement questions and is scored against reference answers. These runs yield the implementation score and clarification score, respectively, without further client guidance. The tasks are based on real software development jobs on Upwork, as detailed in Section 2.4.

2.2 REQUIREMENT CLARIFICATION

The clarification phase determines what information the coding agent will have when it starts development. The agent inspects the supplied materials and asks the client agent about unclear or missing requirements. Its instructions explicitly require clarification of critical delivery details, such as how the software should be invoked and its expected inputs and outputs. The client agent responds using the complete requirements, confirming or correcting the agent's proposed interpretations and answering broader questions within their requested scope.

Because client replies guide both subsequent evaluations, we check them before they reach the coding agent. A supervisor agent reviews each draft for accuracy, completeness relative to the question, and whether it stays within the requested scope. When a draft needs revision, the supervisor explains the problem to the client agent, which revises the reply and submits it for review again. Only the approved reply reaches the coding agent; drafts and review feedback remain hidden.

The coding agent uses each approved reply to decide what to inspect or ask next, or whether to end clarification. One round consists of a question and its approved reply, regardless of the number of internal revisions. The agent may inspect the workspace during this phase but must wait until clarification ends to implement the software. This gives it multiple opportunities to resolve missing details early, while requiring subsequent development to proceed without further client guidance.

2.3 INDEPENDENT EVALUATION

Both evaluations should start from the information available when clarification ends. However, an evaluation question may itself reveal a missing requirement; asking it in the same run before implementation could therefore change the software the agent produces. We avoid this by freezing the clarification context—the initial request, approved conversation, and information obtained from inspecting the workspace—and using it to start two independent runs. Their later messages and results remain separate, preventing assessment questions from guiding code implementation.

Implementation score. The first run tests whether the coding agent can turn the information it acquired into working software. The agent edits files, runs its own checks, and submits the implementation without access to further client replies, private acceptance tests, or reference implementations.

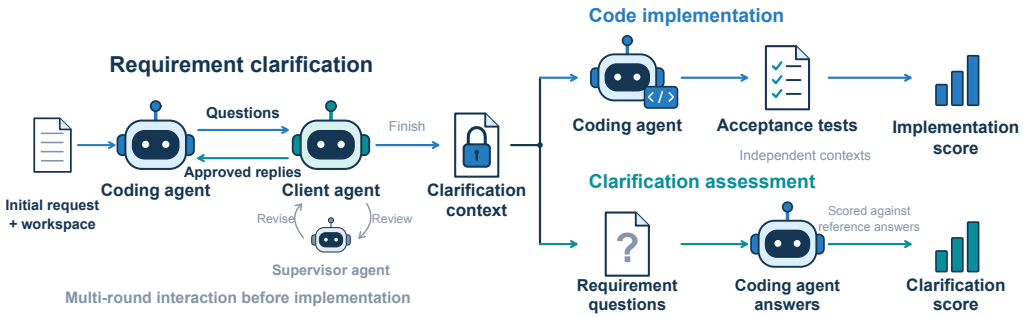

Figure 2: SWE-Clar separates client interaction from two independent evaluations. The coding agent interacts with a client agent whose replies are reviewed by a supervisor agent. The resulting clarification context initializes both branches. Acceptance tests determine the implementation score; answers to requirement questions determine the clarification score. Assessment questions, answers, and scores do not enter the implementation branch.

Acceptance tests then run the submitted software and check its behavior. A task receives 1 if all tests pass and 0 otherwise. The implementation score is the percentage of tasks that pass, equivalent to task pass rate, and serves as our primary outcome.

Clarification score. The second run tests whether the coding agent can correctly explain the requested behavior. The agent answers requirement questions using the same clarification context and read-only task materials. It cannot execute commands, modify files, ask further client questions, or see reference answers. A judge agent compares its answers with the reference answers, without using the implementation's test results. Correct answers establish all material requested behavior without unresolved material contradictions and receive 1. Partial answers establish substantive correct behavior but leave a material omission or uncertainty, without an explicit material contradiction, and receive 0.5. Incorrect answers contain a material contradiction or provide no substantive correct answer and receive 0. Equivalent wording and correct inferences are accepted. The clarification score (%) averages question scores within each task, then equally across tasks.

These scores separate the agent's ability to explain the requirements from its ability to implement them. For additional analysis, we also link each question to acceptance checks for the same requirement. Appendix B.3 defines these requirement units (RUs), the auxiliary RU pass rate, and the answer-grading rubric. Table 3 in Appendix B.1 summarizes information access and permitted actions; Appendix F provides the role prompts.

2.4 Benchmark Construction

The workflow requires tasks for which the initial request is incomplete but the intended software can still be specified and tested. We construct 100 such tasks based on real software development jobs on Upwork, each with at least US$4,000 in total payments. For each commission, GPT-6 Astra analyzes the initial request and recorded client–freelancer conversations to identify requirements critical to delivery, including explicit statements and requirements inferred from context. We preserve the original business goal while replacing dependencies on client accounts and external deployment, allowing local development and evaluation. Appendix A details the task construction process.

The requirements identified in this process determine what the client agent knows. The coding agent receives an incomplete initial request and a workspace containing permitted files, data, and the runtime environment. These materials provide partial requirement information, not a complete checklist; the agent clarifies remaining details through inspection and client interaction. The complete requirements remain private and guide client replies, keeping the task fixed throughout the interaction.

The same requirements also provide the basis for both evaluations. We construct 512 questions paired with reference answers, covering the required behavior and its relevant conditions and exceptions. Separately, acceptance tests check whether the delivered software satisfies those requirements. For the illustrative commission in Figure 1, a question could ask which cancellations qualify for a

refund, while tests would check the program's handling of approved and unapproved cancellations. The question checks the agent's understanding; the tests check the software's behavior.

Running the same tests on different implementations requires agreed delivery interfaces: how to invoke the software, what inputs it accepts, and what outputs it produces. We fix these interfaces together with the questions, reference answers, and tests before evaluation. The coding agent obtains the necessary interface details from the task materials and client interaction, while remaining free to choose the internal implementation.

Two reviewers with relevant industry backgrounds cross-checked the tasks for consistency among client requirements, requirement questions, and acceptance tests.

2.5 Dataset Analysis and Discussion

Task coverage. The 100 tasks span eight categories of development content and several runtime environments, with 512 requirement questions and 3–7 questions per task. Figure 3 shows the number of tasks in each category. This variety allows SWE-Clar to examine clarification and implementation across different types of greenfield projects.

Source conversations. The source conversations also show when requirements missing from the initial request become explicit. We identify their first disclosure and normalize time between each conversation's first and last visible messages. Among conversations with later-disclosed requirements, an average of 69.2% appear within the first quarter of the observed period (Figure 4). This concentration supports examining early clarification, although conversation timestamps alone do not establish when implementation began or whether clients remained available afterward.

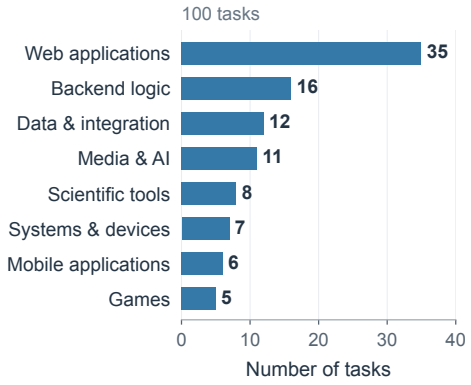

Figure 3: The 100 tasks grouped by development type. Bars are ordered by task count, with counts shown at their ends.

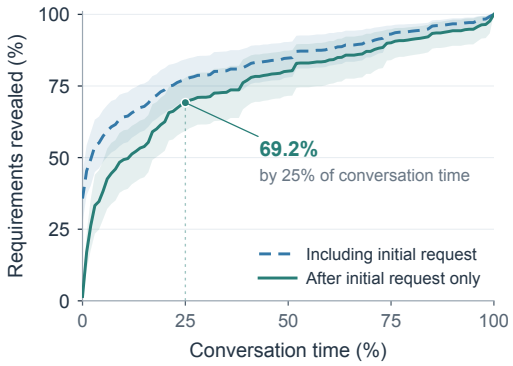

Figure 4: Requirements revealed as client conversations progress. Time runs from the first to the last recorded message. Shading shows pointwise 95% confidence intervals from conversation-level bootstrap resampling.

Comparison with existing benchmarks. Table 1 distinguishes SWE-Clar's task setting and evaluation from representative benchmarks. SWE-Lancer already evaluates paid freelance work, but its coding tasks modify an existing codebase without client clarification (Miserendino et al., 2025). ClarifyCodeBench and Ambig-SWE study clarification for programming problems and repository issues (Fang et al., 2026; Vijayvargiya et al., 2026), while ICAE-Bench extends interaction to from-scratch repository generation (Peng et al., 2026). SWE-Together and SWE-Interact assess development with continued user feedback and report interaction or goal-coverage diagnostics (Wu et al., 2026; Raghavendra et al., 2026). SWE-Clar instead focuses on independently completing real, high-value greenfield commissions after early client interaction. Its separate question-answering run directly checks which requirements the agent understands, while acceptance tests check what it delivers. Evaluating both from the same clarification context makes these outcomes comparable without letting assessment questions influence the software.

Table 1: Representative coding benchmarks compared by task setting, client interaction, and evaluation. Independent requirement QA scores the coding agent's answers in a context isolated from code implementation. SWE-Lancer refers to its individual-contributor coding tasks.

Benchmark	Task setting	Client interaction	Evaluation
SWE-Lancer (2025)	Upwork commissions; existing-codebase changes	No client clarification	End-to-end code tests
ClarifyCodeBench (2026)	LiveCodeBench; programming problems	Questions before code generation	Question quality and efficiency; code tests
Ambig-SWE (2026)	SWE-bench Verified; issue resolution	Queries to a simulated issue reporter	Information recovery; patch tests
SWE-Together (2026)	Recorded coding sessions; repository tasks	Feedback throughout development	Intent coverage, user correction, and repository correctness
SWE-Interact (2026)	Existing SWE benchmarks; repository tasks	Progressive disclosure and implementation feedback	Plan/code goal coverage; task verifiers
ICAE-Bench (2026)	Open-source repositories; from-scratch generation	Requirement queries during project building	Constraint coverage; black-box code tests
SWE-Clar	High-value Upwork commissions; greenfield projects	Reviewed clarification before implementation	Independent requirement QA; code acceptance

3 EXPERIMENTS

In this section, we evaluate coding agents on SWE-Clar to answer three research questions:

- **RQ1:** How well do coding agents clarify requirements and implement greenfield software?
- **RQ2:** How much does clarification help when the initial requirements are incomplete?
- **RQ3:** How does requirement-answer correctness relate to implementation success?

3.1 EXPERIMENTAL SETUP

Tasks and metrics. We assign each model the same 100 SWE-Clar tasks and report the two scores defined in Section 2.3: **implementation score**, the percentage of tasks that pass all acceptance tests, and **clarification score**, the average requirement-question score. Both are reported as percentages. Table 2 presents the main comparison.

Models and protocol. We evaluate GPT-6 Astra, GPT-5.6 Sol, GPT-5.6 Luna, Claude Fable 5.1, Claude Opus 5, GLM-5.3, GLM-5.3 Flash, Kimi K3, and Qwen-3.8 Max as coding agents. We use the highest supported reasoning effort for each coding model. The standard protocol uses GPT-5.6 Luna as the client agent, supervisor agent, and judge agent. Each coding agent completes clarification before the two evaluation branches begin. Appendix B.1 describes role configuration and information access.

Comparisons and uncertainty. We first compare models under the standard protocol, then examine the effects of three requirement conditions and compare answers with code-test outcomes for the same requirements. For implementation-score differences between requirement conditions, we report 95% confidence intervals (Appendix C.2). Additional analyses examine completion cost, requirement disclosure across interaction rounds, and detailed answer–implementation outcomes and cases (Appendix C), followed by clarification timing (Appendix D) and reply quality before and after Supervisor review (Appendix E).

3.2 Overall Performance (RQ1)

We begin by examining how well the models clarify client requirements and complete the resulting software tasks. Table 2 reports both scores. Two observations follow:

- **Complete software delivery remains challenging.** Implementation scores range from 39.0% to 67.0%. GPT-6 Astra achieves the highest observed score but still fails 33 of the 100 tasks.
- **Higher clarification scores accompany higher implementation scores.** Across the nine models, the two scores show a strong positive association (Pearson $r = 0.89$). RQ3 examines this relationship further at the level of individual requirements.

Table 2: SWE-Clar evaluation on 100 tasks per model. Bold and underlined values indicate the highest and second-highest scores; higher is better. Brackets show 95% task-bootstrap confidence intervals (10,000 resamples).

Coding model	Implementation score (%)	Clarification score (%)
GPT-6 Astra	**67.0** [58, 76]	**87.5** [85.0, 90.1]
Claude Opus 5	<u>63.0</u> [53, 72]	84.0 [80.9, 86.8]
Claude Fable 5.1	60.0 [50, 69]	<u>84.3</u> [81.0, 87.3]
GPT-5.6 Sol	57.0 [47, 67]	78.1 [74.9, 81.3]
Qwen-3.8 Max	55.0 [45, 64]	81.3 [77.9, 84.7]
GLM-5.3	54.0 [44, 64]	76.8 [73.3, 80.2]
Kimi K3	49.0 [39, 59]	76.9 [73.4, 80.3]
GPT-5.6 Luna	49.0 [39, 59]	71.8 [68.6, 74.9]
GLM-5.3 Flash	39.0 [30, 49]	73.2 [68.9, 77.3]

3.3 Effect of Clarification (RQ2)

We next examine how access to client requirements changes both scores. For GPT-5.6 Luna and GLM-5.3 Flash, we compare *no client interaction*, the *standard clarification protocol*, and *complete requirements provided upfront*. The task workspace is retained, and complete requirements do not reveal private tests or reference implementations. Both models are compared on the same 100 tasks under each condition. Requirement answers in all three conditions are evaluated with the same grading rubric used in the main comparison. Figure 5 presents the results; paired differences and intervals are reported in Appendix C.2.

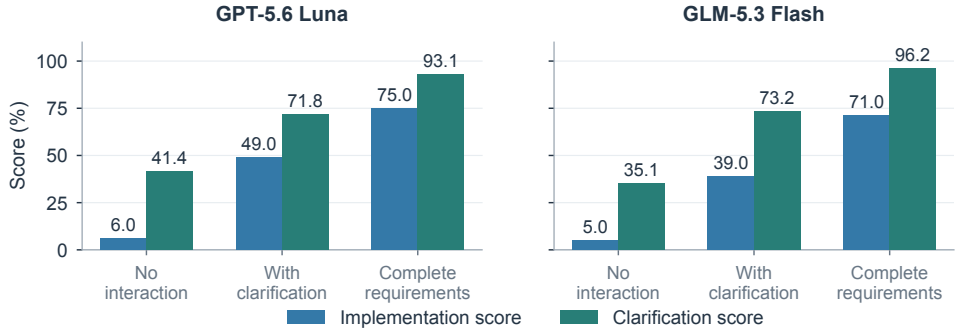

Figure 5: Effect of requirement availability on the two scores. Each model is compared on 100 tasks under each condition. The same grading rubric is used across conditions and in the main comparison.

- **Clarification improves both requirement understanding and software delivery.** With client interaction, GPT-5.6 Luna's implementation score rises from 6.0% to 49.0%, while GLM-5.3 Flash improves from 5.0% to 39.0%. Their clarification scores also increase by 30.4 and 38.1 percentage points, respectively. These results show improvements in both answering requirement questions and completing the software.

- **Clarification recovers only part of the gap to complete requirements.** Relative to no interaction, the standard protocol closes 62.3% of the implementation-score gap for GPT-5.6 Luna and 51.5% for GLM-5.3 Flash. Complete requirements further raise implementation scores to 75.0% and 71.0%, respectively. Missing information thus remains consequential, while the unsolved tasks under complete requirements show that implementation itself is also difficult.

3.4 Requirement Answers and Implementation (RQ3)

The preceding results compare task-level performance across models. We next examine how requirement-answer correctness relates to implementation success. Each task contains multiple requirement units (RUs), each representing a specific client requirement with its own requirement question and acceptance tests (Appendix B.3). Each model is evaluated on all 512 RUs across 100 tasks. The joint analysis pairs the answer judgment for each of these 512 RUs with its corresponding code-test outcome. Figure 6 shows results for correct, partial, and incorrect answers. Appendix C.5 provides the full joint distribution.

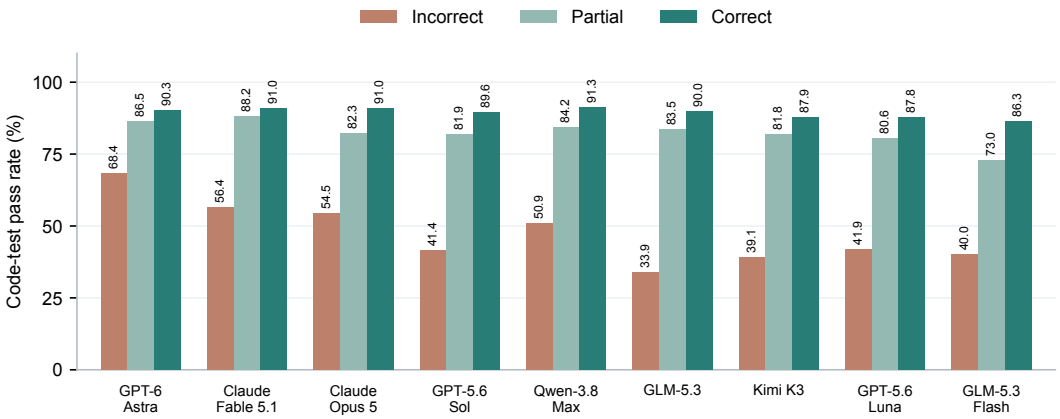

Figure 6: Code-test pass rates within answer categories. Each bar uses the RUs in that model's answer category as its denominator. Incorrect includes contradictory or substantively unanswered requirements.

Incorrect answers identify requirements more likely to fail implementation. Across models, code-test pass rates are 33.9–68.4% for incorrect answers, compared with 86.3–91.3% for correct answers and 73.0–88.2% for partial answers. Correct answers have higher pass rates than partial answers in all nine models. Answer correctness is therefore associated with implementation success, while the two measurements capture different aspects of task performance.

Requirement answers and code tests provide complementary evidence. The model-level correlation does not imply agreement on individual requirements: correct answers with failing implementations account for 6.1–7.8% of all 512 RUs per model. Partial answers establish substantive correct behavior but leave a material omission or uncertainty without an explicit material contradiction. An unstated detail need not be absent from the code. For example, in the filtering case in Appendix C.5.1, the answer omits the required empty-result behavior when no items match, but the implementation passes the corresponding test, which includes this case. Conversely, correct answers can accompany failing implementations, while incorrect answers can accompany passing implementations. Requirement questions thus measure what agents can explain, whereas acceptance tests check what they implement; the two outcomes are related but not interchangeable.

Disclosure does not guarantee correct answers. After matching client-disclosure records to requirement answers from the same runs, 2.8–6.4% of the initially hidden RUs fully disclosed by the end of clarification receive Incorrect labels across models (Appendix C.4). This distinguishes acquiring requirement information from understanding it correctly.

4 Related Work

Software Development Benchmarks. Coding benchmarks have expanded from generating individual functions to completing software engineering tasks. HumanEval evaluates generated functions with executable tests (Chen et al., 2021), while SWE-bench asks agents to resolve GitHub issues in existing repositories (Jimenez et al., 2024). SWE-Lancer further connects evaluation to paid freelance work through Upwork tasks from the Expensify codebase and their historical payouts (Miserendino et al., 2025). Another line of work evaluates complete repository generation: DevEval supplies product requirements and software designs, while NL2Repo-Bench provides natural-language specifications and evaluates generated repositories against upstream tests (Li et al., 2025; Ding et al., 2025). These benchmarks establish the importance of evaluating working software in realistic settings. SWE-Clar focuses on greenfield software development in freelance settings, where agents must build new software from incomplete client requests. Its tasks preserve real business goals and require clarification before independent delivery.

Requirement Clarification. Incomplete requests require agents to determine what information is missing and obtain it before making consequential implementation choices. ClarifyGPT identifies ambiguous requirements and asks targeted questions to refine them before code generation (Mu et al., 2024). At the programming-problem level, HumanEvalComm and ClarifyCodeBench evaluate whether models ask useful clarification questions, with ClarifyCodeBench additionally measuring interaction efficiency (Wu & Fard, 2025; Fang et al., 2026). At the repository level, Ambig-SWE examines how agents recover information omitted from issue descriptions (Vijayvargiya et al., 2026). ICAE-Bench extends interactive evaluation to from-scratch repository generation, deriving incomplete requirements and grounded replies from open-source projects (Peng et al., 2026). SWE-Clar studies clarification in tasks based on real software development jobs on Upwork. Agents clarify requirements through multiple exchanges, then implement the software without further client guidance.

Interaction and Process Evaluation. Evaluating interactive agents involves both the user simulation and the evidence used to assess performance. τ-bench combines simulated users with tool-using agents and checks task outcomes against the resulting environment state (Yao et al., 2025). In coding, SWE-Together builds tasks from real user sessions and measures repository correctness and user correction, while its intent-coverage metric assesses the simulator's fidelity to the original user requests (Wu et al., 2026). SWE-Interact provides feedback during development and examines goal coverage in plans and code revisions (Raghavendra et al., 2026). SWE-RPG uses validated requirement and planning references to assess coding trajectories and attribute failures to requirement, planning, or coding stages (Zhou et al., 2026). These approaches show the value of examining more than the final code outcome. SWE-Clar obtains explicit evidence of requirement understanding through a separate question-answering run after clarification. Both runs start from the same context, allowing comparison without letting assessment questions guide implementation.

5 Conclusion

In this work, we introduced **SWE-Clar**, a benchmark for requirement clarification and code implementation in greenfield software development in freelance settings. Based on real, high-value software development jobs on Upwork, its tasks require coding agents to clarify incomplete requests before independently developing software. We paired acceptance testing with a separate requirement-question assessment, yielding implementation and clarification scores without allowing assessment questions to guide coding. Evaluations of nine frontier models showed that complete delivery remains challenging. Comparisons across requirement conditions further showed that clarification improves performance but recovers only part of the gap to complete requirements, while model-level score correlations coexist with requirement-level answer–code disagreements, demonstrating the complementary information provided by the two evaluations.

Future work can extend SWE-Clar to changing requirements and different patterns of client availability, and investigate how coding agents can clarify client needs more effectively and carry them through to reliable software delivery.

AI USE STATEMENT

Generative AI tools assisted with manuscript translation and revision, literature research, interpretation of the reported results, and figure preparation, including plotting and layout code. GPT-6 Astra assisted with source-material processing, task construction, and development of requirement questions, acceptance tests, and reference implementations, with human cross-validation as described in Section 2.4 and Appendix A. LLM use in coding, client simulation, reply review, and answer grading is documented in the method, experiments, and appendix. We take full responsibility for verifying AI-assisted text, references, benchmark materials, and figures, and for all final claims and artifacts.

REPRODUCIBILITY STATEMENT

The supplementary material provides the 100 tasks, 512 requirement units, acceptance tests, role prompts, evaluation harness, and environment build configurations. It also includes reference, behaviorally equivalent, and targeted incorrect implementations for each task. The appendix describes evaluation and scoring, and the supplement includes integrity checks for the task materials and implementation files.

REFERENCES

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.

Jingzhe Ding, Shengda Long, Changxin Pu, Huan Zhou, Hongwan Gao, Xiang Gao, Chao He, Yue Hou, Fei Hu, Zhaojian Li, Weiran Shi, Zaiyuan Wang, Daoguang Zan, Chenchen Zhang, Xiaoxu Zhang, Qizhi Chen, Xianfu Cheng, Bo Deng, Qingshui Gu, Kai Hua, Juntao Lin, Pai Liu, Mingchen Li, Xuanguang Pan, Zifan Peng, Yujia Qin, Yong Shan, Zhewen Tan, Weihao Xie, Zihan Wang, Yishuo Yuan, Jiayu Zhang, Enduo Zhao, Yunfei Zhao, He Zhu, Liya Zhu, Chenyang Zou, Ming Ding, Jianpeng Jiao, Jiaheng Liu, Minghao Liu, Qian Liu, Chongyang Tao, Jian Yang, Tong Yang, Zhaoxiang Zhang, Xinjie Chen, Wenhao Huang, and Ge Zhang. NL2Repo-Bench: Towards long-horizon repository generation evaluation of coding agents. *arXiv preprint arXiv:2512.12730*, 2025.

B. Efron. Bootstrap methods: Another look at the jackknife. *The Annals of Statistics*, 7(1):1–26, 1979.

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.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *International Conference on Learning Representations (ICLR)*, 2024.

Brendan King and Jeffrey Flanigan. Dialogue SWE-Bench: A benchmark for dialogue-driven coding agents. *arXiv preprint arXiv:2606.13995*, 2026.

Bowen Li, Wenhan Wu, Ziwei Tang, Lin Shi, John Yang, Jinyang Li, Shunyu Yao, Chen Qian, Binyuan Hui, Qicheng Zhang, Zhiyin Yu, He Du, Ping Yang, Dahua Lin, Chao Peng, and Kai Chen. Prompting large language models to tackle the full software development lifecycle: A case study. In *Proceedings of the 31st International Conference on Computational Linguistics*, pp. 7511–7531. Association for Computational Linguistics, 2025. URL https://aclanthology.org/2025.coling-main.502/.

Mantas Mazeika, Alice Gatti, Cristina Menghini, Udari Madhushani Sehwag, Shivam Singhal, Yury Orlovskiy, Steven Basart, Manasi Sharma, Denis Peskoff, Elaine Lau, Jaehyuk Lim, Lachlan Carroll, Alice Blair, Vinaya Sivakumar, Sumana Basu, Brad Kenstler, Yuntao Ma, Julian Michael, Xiaoke Li, Oliver Ingebretsen, Aditya Mehta, Jean Mottola, John Teichmann, Kevin Yu, Zaina Shaik, Adam Khoja, Richard Ren, Jason Hausenloy, Long Phan, Ye Htet, Ankit Aich, Tahseen

Rabbani, Vivswan Shah, Andriy Novykov, Felix Binder, Kirill Chugunov, Luis Ramirez, Matias Geralnik, Hernán Mesura, Dean Lee, Ed-Yeremai Hernandez Cardona, Annette Diamond, Summer Yue, Alexandr Wang, Bing Liu, Ernesto Hernandez, and Dan Hendrycks. Remote labor index: Measuring AI automation of remote work. *arXiv preprint arXiv:2510.26787*, 2025.

Samuel Miserendino, Michele Wang, Tejal Patwardhan, and Johannes Heidecke. SWE-Lancer: Can frontier LLMs earn $1 million from real-world freelance software engineering? In *International Conference on Machine Learning (ICML)*, volume 267 of *Proceedings of Machine Learning Research*, pp. 44412–44450, 2025.

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.

Bashar Nuseibeh and Steve Easterbrook. Requirements engineering: A roadmap. In *Conference on the Future of Software Engineering*, pp. 35–46. ACM, 2000.

Sida Peng, Eirini Kalliamvakou, Peter Cihon, and Mert Demirer. The impact of AI on developer productivity: Evidence from GitHub Copilot. *arXiv preprint arXiv:2302.06590*, 2023.

Zhongyuan Peng, Dan Huang, Chuyu Zhang, Caijun Xu, Changyi Xiao, Shibo Hong, David Lo, Lin Qiu, Xuezhi Cao, Jiyuan He, and Yixin Cao. ICAE-Bench: Evaluating coding agents as interactive project builders. *arXiv preprint arXiv:2607.21217*, 2026.

Mohit Raghavendra, Anisha Gunjal, Aakash Sabharwal, and Yunzhong He. SWE-INTERACT: Reimagining SWE benchmarks as user-driven long-horizon coding sessions. *arXiv preprint arXiv:2606.30573*, 2026.

Quan Shi, Keshav Dhandhania, Karthik Narasimhan, and Victor Barres. τ^{τ}-Bench: An environment for end-to-end, realistic agent construction. *arXiv preprint arXiv:2609.04611*, 2026.

Tu Trinh, Mohamed Elfeki, Guangze Luo, Kelvin Luu, Nathan Hunt, Ernesto Hernandez, Nandan Marwaha, Yannis Yiming He, Charles Wang, Fernando Carabedo, Alessa Castillo, and Bing Liu. HiL-Bench (human-in-loop benchmark): Do agents know when to ask for help? *arXiv preprint arXiv:2604.09408*, 2026.

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 (ICLR)*, 2026.

Jie JW Wu and Fatemeh H Fard. HumanEvalComm: Benchmarking the communication competence of code generation for LLMs and LLM agent. *ACM Transactions on Software Engineering and Methodology*, 2025. doi: 10.1145/3715109.

Yifan Wu, Zhuokai Zhao, Songlin Li, Ho Hin Lee, Jiacheng Zhu, Shirley Wu, Tianhe Yu, Serena Li, Lizhu Zhang, Xiangjun Fan, and Shengzhi Li. SWE-Together: Evaluating coding agents in interactive user sessions. *arXiv preprint arXiv:2606.29957*, 2026.

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 (NeurIPS)*, volume 37, pp. 50528–50652, 2024.

Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. τ-bench: A benchmark for tool-agent-user interaction in real-world domains. In *International Conference on Learning Representations (ICLR)*, 2025.

Darvin Yi, Teng Liu, Mattie Terzolo, Lance Hasson, Ayan Sinha, Pablo Mendes, and Andrew Rabinovich. UpBench: A dynamically evolving real-world labor-market agentic benchmark framework built for human-centric AI. *arXiv preprint arXiv:2511.12306*, 2025.

Xin Zhou, Chun Yong Chong, Kisub Kim, Yun Peng, Rui Shu, Zihan Wu, Xu Han, Guowen Yuan, Zeyang Zhuang, Jounghoon Kim, Jeongjin Ju, Seongmin Ju, Taein Yoon, and David Lo. A unified issue resolution benchmark for requirement clarification, planning, and code generation for coding agents. *arXiv preprint arXiv:2608.09072*, 2026.

A TASK CONSTRUCTION

Section 2.4 describes task construction. This appendix adds details on material handling and the analysis of requirement disclosure in source conversations.

We first identify 122 candidate jobs with at least eight milestones, at least US$4,000 in total payments, and at least 20 messages. We select 100 jobs that can be reproduced locally, can avoid dependencies on external client accounts, and have deliverables suitable for evaluation through acceptance tests. Design-oriented tasks that are difficult to evaluate objectively are excluded. The monetary threshold refers to total payments for the source job.

GPT-6 Astra first processes the source materials. Two reviewers with relevant industry backgrounds then cross-check the tasks for consistency among client requirements, requirement questions, and acceptance tests. GPT-6 Astra and human contributors jointly develop the acceptance tests and reference implementations. All reference implementations pass every acceptance test for their respective tasks.

The supplied starter projects provide structure and, in some tasks, runnable baseline functionality; agents must complete the required target behavior. Greenfield refers to the new software projects underlying the tasks, rather than an empty starting directory. None of the 512 RUs across the 100 tasks passes its corresponding acceptance tests in the initial state, giving a starter RU pass rate of 0%. Local workspaces use synthetic records and controlled services in place of original client accounts and deployment environments, allowing the agent to develop the required software locally. The release contains the curated benchmark tasks and evaluation harness, not original client conversations or identifying information.

We further check the acceptance tests using behaviorally equivalent implementations and targeted incorrect implementations. Across the 100 tasks, both reference and equivalent implementations pass all acceptance tests, while an incorrect implementation targeting one requirement per task is rejected by the corresponding RU tests. These checks verify acceptance of correct alternative implementations and detection of the tested requirement violations. The supplementary validation materials include the three implementation variants for every task and their recorded test outcomes.

A.1 PRIVACY PROTECTION AND RELEASE SCOPE

Real client commissions inform the business goals and requirements of the benchmark. The dataset and evaluation code are included in the supplementary material, which excludes original conversations, client identities, source-job links, and conversation excerpts. Tasks use benchmark-specific identifiers without a public mapping to the original clients or commissions.

Tasks use synthetic records and controlled local services in place of original business records, client accounts, and deployment environments. The dataset contains no real client identities or contact information; fields needed for testing use synthetic data or placeholders. Evaluation does not require access to real client accounts. Before release, we inspect code, task materials, file paths, and media metadata for identifying information and credentials, while preserving third-party licenses and copyright notices. The paper reports aggregate statistics from source-conversation analysis without releasing individual conversation contents.

A.2 REQUIREMENT DISCLOSURE IN SOURCE CONVERSATIONS

We analyze requirements in the initial request and client interaction, merge equivalent repetitions, and use the client's final confirmed requirements when earlier requirements change. We identify when these requirements become explicit in the conversation to analyze the timing of requirement disclosure. Unconfirmed freelancer proposals, scheduling, payment updates, and access logistics are excluded. The public dataset does not include supporting excerpts or message positions from the original conversations.

For each conversation, elapsed time is normalized between its first and last visible messages. Initial-request requirements are placed at time zero. The primary curve divides the cumulative number of later-disclosed requirements by the total number of such requirements observed in that conversation; conversations without later-disclosed requirements do not enter this mean. The supplementary curve

includes initial-request requirements in both numerator and denominator. Figure 4 averages these curves equally across conversations; its shading shows pointwise 95% confidence intervals obtained from 2,000 conversation-level bootstrap resamples.

B Evaluation Protocol and Scoring

B.1 Roles, Context, and Tools

Our coding-agent runtime is built on Pi.[2] Table 3 specifies information access and permitted actions for the procedure in Sections 2.2 and 2.3. The standard protocol sets no fixed limit on clarification rounds or question count. The coding agent autonomously decides when to end each phase: it calls `finish_clarification` to end clarification and `submit_delivery` to submit its implementation and end coding; Appendix F provides the full prompts.

The saved model configurations set reasoning effort to `max`. Where `max_tokens` is unspecified, response length follows the corresponding service and model defaults rather than a common total-token budget. Acceptance tests directly determine implementation scores; the judge agent is used only to grade requirement answers.

Table 3: Agent responsibilities and information access. Both evaluation branches start from the frozen clarification context. Private requirement materials are accessible only to client simulation and evaluation agents.

Phase or branch	Agent	Available information	Actions and output
Requirement clarification	Coding	Initial request, permitted workspace, approved conversation	Inspect materials, ask questions, decide when to stop
	Client	Complete requirements, task materials, approved conversation, current question	Draft or revise a reply within the requested scope
	Supervisor	Same information as the Client, plus the draft reply	Approve the reply or request revision
Code implementation	Coding	Frozen context and development workspace	Implement, run development checks, submit software for acceptance testing
Clarification assessment	Coding	Frozen context, requirement questions, read-only development materials	Answer questions without further interaction or implementation
	Judge agent	Reference requirements and answers, candidate answers, relevant acceptance tests	Grade requirement answers independently of code outcomes

B.2 Timeouts and Failure Handling

The released harness defaults to a 900-second timeout without model or tool progress. This limits inactivity rather than total task duration. Coding has no total-duration limit by default; runtime configuration can override these limits.

Retryable execution errors are handled through saved workflow checkpoints, with at most five automatic retries at the same checkpoint. Exhausting this limit stops the run and records a failure. A retryable evaluation error triggers reevaluation of the submitted implementation without recoding. A completed delivery is not automatically retried because of a low score. Execution errors and acceptance-test failures are recorded separately.

[2] `https://github.com/earendil-works/pi`

B.3 Requirement-Level Scoring

Requirement units (RUs) link each client requirement to its question and acceptance tests. The auxiliary RU pass rate averages the proportion of passing RUs within each task and then equally across tasks.

For each RU, the judge agent evaluates the answers against the required behavior and agreed delivery interface. Corresponding acceptance tests clarify how these requirements translate into software behavior without introducing additional requirements. The judge examines all answers for the same task: relevant evidence in an answer to another RU also counts and need not be repeated in the target answer. Equivalent wording is accepted, but generic commitments such as "follow the requirements" do not establish knowledge of specific requirements. The judge evaluates the answers independently of code-test outcomes.

Requirement questions use consistent templates to identify the topic being assessed; answers must still specify the concrete behavioral rules in the reference requirements. Questions are provided only to the independent answer branch after the clarification context is frozen and do not enter the coding branch. This prevents the coding agent from inferring additional requirements from the assessment questions, preserving the evaluation of independent implementation based on the clarification context.

Requirement-answer grading uses Correct (1), Partial (0.5), and Incorrect (0). The judge evaluates answers against the requirements, delivery interface, and corresponding test source, without access to code-test outcomes; tests cannot introduce undeclared requirements. Verbatim evidence quotations are not required. Reference conflicts are resolved through repeated review or requirement-based manual adjudication, with the original records retained. Manual checks confirmed agreement between the judge's labels and human judgments on the reviewed samples.

Table 4: Requirement-answer grading rubric.

Label	Criterion	Score
Correct	States all material requirements correctly.	1
Partial	Establishes substantive correct behavior, with a material omission or uncertainty and no explicit material contradiction.	0.5
Incorrect	States incompatible behavior or provides no substantive correct answer.	0

C Additional Experimental Results

This appendix supplements the main results with cost and RU pass rates, detailed comparisons of information conditions, disclosure during clarification, and full answer–implementation outcomes.

C.1 Overall Performance and Cost

Cost covers the coding agent's calls during clarification and code implementation. Client, Supervisor, requirement-answering, and judge-agent calls are excluded. We compute cost from the saved input, output, and cache usage, priced at the actual recorded rates; we use the five-minute cache-write rate when the cache lifetime is unspecified.

Table 5 summarizes mean cost and supplementary RU pass rate for 100 tasks per model. Figure 7 compares estimated mean cost with the 100-task implementation scores.

Higher call cost does not consistently correspond to higher task completion: Astra has a lower mean cost than either Claude model but a higher observed implementation score. Across models, RU pass rates exceed whole-task pass rates, distinguishing the ability to implement many individual requirements from the ability to satisfy all requirements of a task together.

Table 5: Supplementary RU pass rates and mean costs for 100 evaluation tasks per model.

Model	RU pass (%)	USD/task
GPT-6 Astra	88.5	5.391
Claude Fable 5.1	86.7	11.033
Claude Opus 5	87.2	9.694
GPT-5.6 Sol	84.4	1.965
Qwen-3.8 Max	85.9	1.703
GLM-5.3	82.4	1.393
Kimi K3	82.2	1.338
GPT-5.6 Luna	82.7	0.103
GLM-5.3 Flash	77.8	0.095

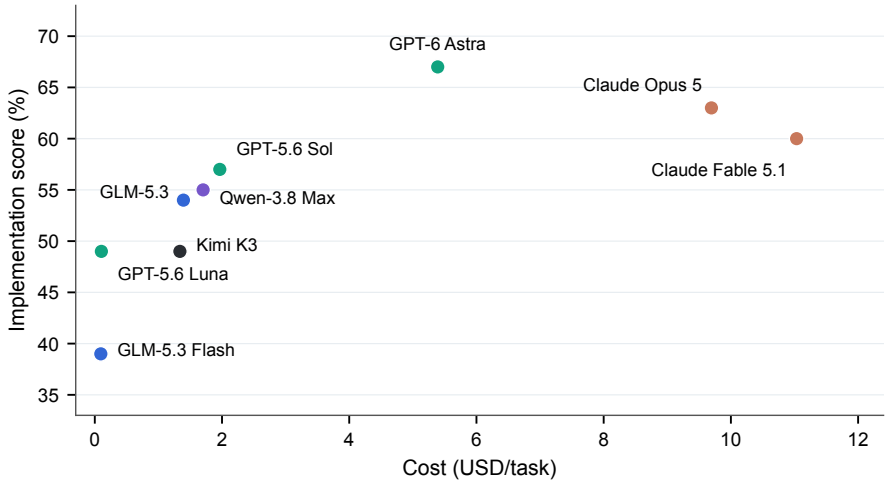

Figure 7: Cost and implementation score. Cost includes coding-agent calls during clarification and implementation.

C.2 Information Availability

For the information conditions in Section 3.3, Table 6 adds RU pass rates and exact scores, while Table 7 reports paired implementation-score differences. The 95% confidence intervals use 10,000 task-level bootstrap resamples, selecting the same task IDs across conditions in each resample (Efron, 1979).

Table 6: Information conditions on matched tasks within each candidate. Scores are percentages.

Candidate	Information condition	Implementation score	Clarification score	RU pass rate
GPT-5.6 Luna	No client interaction	6.0	41.4	37.5
GPT-5.6 Luna	Standard protocol	49.0	71.8	82.7
GPT-5.6 Luna	Complete requirements	75.0	93.1	93.2
GLM-5.3 Flash	No client interaction	5.0	35.1	36.0
GLM-5.3 Flash	Standard protocol	39.0	73.2	77.8
GLM-5.3 Flash	Complete requirements	71.0	96.2	91.1

Both models improve from no interaction to the standard protocol and from the standard protocol to complete requirements; all four paired 95% intervals exclude zero. Clarification therefore recovers part of the performance lost under incomplete initial information, while the further gains with complete requirements identify more thorough information acquisition as a promising direction.

Table 7: Paired task-pass-rate differences in percentage points, with task-bootstrap 95% intervals.

Candidate	Comparison	Difference	95% interval
GPT-5.6 Luna	Standard minus no interaction	43.0	[33.0, 53.0]
GPT-5.6 Luna	Complete minus standard	26.0	[14.0, 37.0]
GLM-5.3 Flash	Standard minus no interaction	34.0	[25.0, 43.0]
GLM-5.3 Flash	Complete minus standard	32.0	[20.0, 43.0]

C.3 Requirement Disclosure during Clarification

Using the approved question–answer rounds defined in Section 2.2, we identify when each initially hidden RU is first fully disclosed through model-assisted annotations of client replies. Full disclosure requires all material clauses; partial and uncertain disclosures are not counted as complete, and repeated statements are counted once. The final value is carried forward after a conversation ends normally.

To summarize disclosure in typical conversations, we exclude conversations whose question–answer counts fall below the first quartile minus 1.5 times the interquartile range or above the third quartile plus 1.5 times that range, separately for each model, and average disclosure fractions equally over the retained conversations. The curve records first complete disclosure, so later contradictions do not reduce it; each curve ends at the maximum retained round count for that model. Figure 8 shows the coding agent's progress in obtaining requirement information from the client, reflecting its ability to elicit requirements through clarification under the current client response policy. This ability to acquire requirement information through clarification represents a distinct capability from those measured by clarification and implementation scores.

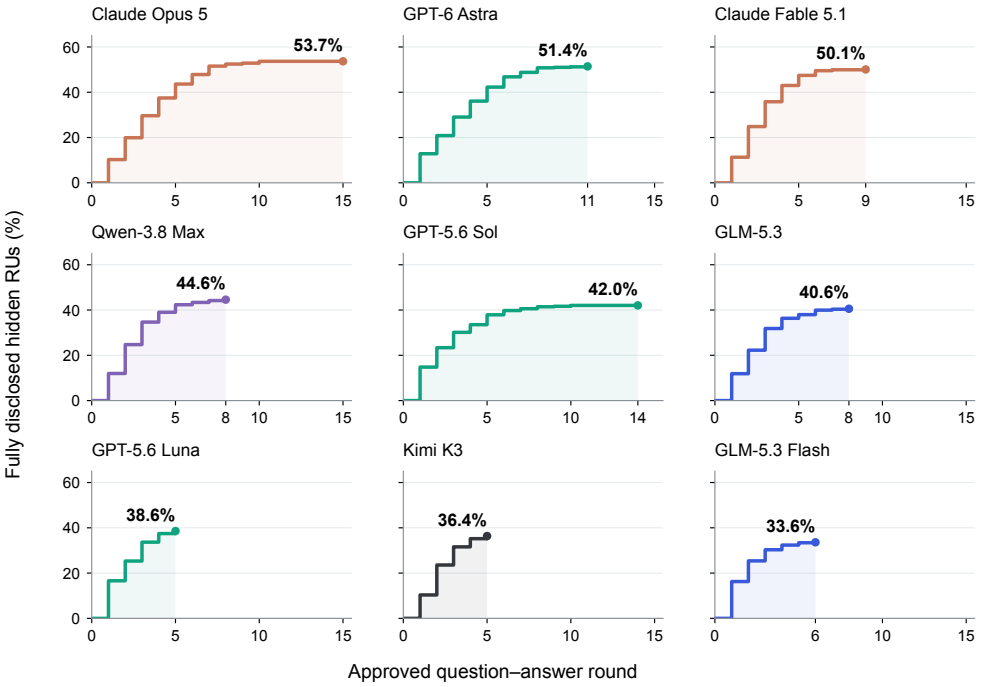

Figure 8: Mean requirement disclosure after filtering conversations with outlying round counts. Models are ordered by their final proportion of fully disclosed hidden RUs. Additional horizontal-axis ticks mark each curve's final round.

After filtering, final mean disclosure ranges from 33.6% to 53.7% of initially hidden RUs across models. Under the current client response policy, agents thus end clarification with a substantial share of hidden requirements not yet fully disclosed, motivating further study of question coverage and stopping decisions.

C.4 Requirement Answers after Full Disclosure

Requirement disclosure measures the extent to which the coding agent elicits complete information about initially hidden requirements through questions to the client; clarification score assesses the agent's understanding of task requirements through requirement questions. The former concerns acquiring the needed information, while the latter concerns understanding it accurately. We therefore examine whether agents correctly answer questions about requirements for which they have obtained complete information through clarification.

To distinguish information acquisition from requirement understanding, we pair disclosure annotations from normally completed clarification conversations with answer judgments from the same runs, without filtering outlying round counts. We select initially hidden RUs annotated as fully disclosed at the end of clarification. Table 8 reports Partial and Incorrect answer rates using the number of eligible RUs for each model as the denominator.

Table 8: Partial and Incorrect answers among fully disclosed hidden RUs. Counts give the eligible RUs per model; remaining answers are Correct.

Model	Fully disclosed RUs	Partial (%)	Incorrect (%)
GPT-6 Astra	249	2.4	3.2
Claude Fable 5.1	247	2.8	4.9
Claude Opus 5	261	3.8	4.6
GPT-5.6 Sol	212	10.4	2.8
Qwen-3.8 Max	220	3.2	6.4
GLM-5.3	202	4.0	6.4
Kimi K3	198	8.1	5.1
GPT-5.6 Luna	201	14.9	3.0
GLM-5.3 Flash	172	4.7	2.9

Even after full disclosure, 2.8%–6.4% of answers are graded Incorrect and another 2.4%–14.9% are Partial across models. Successfully acquiring requirement information therefore does not guarantee accurate understanding, highlighting the value of independent requirement-answer assessment.

The two measures also differ in scope and scoring. Disclosure rate counts the proportion of initially hidden RUs for which the coding agent obtains complete information; clarification score covers all requirement questions and gives half credit to Partial answers. Agents can also reason from the supplied materials. The two percentages therefore should not be directly subtracted, and disclosure rate is not an upper bound on clarification score.

C.5 Requirement Understanding and Implementation

The strict answer-correctness rate gives credit only to fully correct answers and averages tasks equally. Answer-label counts and joint answer/code outcomes instead count individual RU observations. Table 9 reports strict answer correctness and label counts, while Figure 9 reports the full distribution of answer and code outcomes.

Partial answers outnumber Incorrect answers for every model, indicating that incomplete answers are more common than answers graded incorrect. The three-level rubric distinguishes partial from complete knowledge, while strict correctness reveals whether agents can fully understand and articulate requirements.

Figure 9 complements Section 3.4 with the full joint distribution of three answer labels and two code-test outcomes. Whereas Figure 6 compares test-pass rates within answer categories, each cell here uses all 512 RUs for the model as its denominator, showing the prevalence of each outcome in the evaluation.

Successful software delivery requires both clarification and implementation capabilities. Requirement clarification and code implementation are related. As shown in Section 3.4, requirements answered correctly have higher implementation success rates than those answered partially or incorrectly across all nine models. The full joint distribution in Figure 9 further shows that the two

Table 9: Strict answer correctness and answer-label counts over 100 tasks per model. Each model has 512 requirement answers.

Model	Strict (%)	Correct	Partial	Incorrect
GPT-6 Astra	79.0	404	89	19
Claude Fable 5.1	76.1	388	85	39
Claude Opus 5	76.3	389	79	44
GPT-5.6 Sol	62.2	317	166	29
Qwen-3.8 Max	74.0	379	76	57
GLM-5.3	64.7	329	127	56
Kimi K3	63.2	323	143	46
GPT-5.6 Luna	49.9	254	227	31
GLM-5.3 Flash	58.1	293	159	60

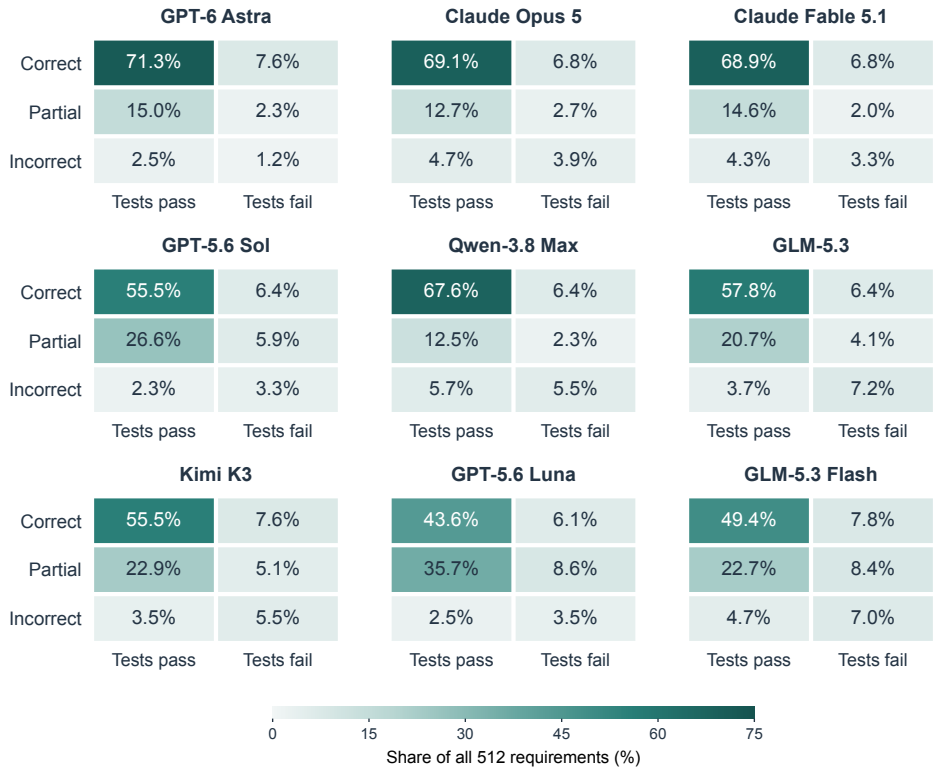

Figure 9: Joint distribution of requirement answers and code-test outcomes. Cells show percentages of all 512 RUs per model; the six cells in each panel sum to 100%, up to rounding. All panels share one color scale and follow the main-table model order. Tests pass and Tests fail refer to acceptance tests for the corresponding RU.

capabilities do not always align: agents may correctly understand and articulate requirements yet fail to implement them, or pass the corresponding tests without fully demonstrating their understanding in the answers. Understanding established during clarification thus provides an important foundation for implementation, while delivery also depends on the agent's ability to translate requirements into code.

Agent development should improve both clarification quality and the translation of requirements into code. One possible explanation for incomplete answers accompanying successful implementations is that agents further infer the required behavior during coding by combining available

materials, implementation details, and their own checks. Conversely, correct answers accompanying failed implementations suggest that accurate requirement understanding may be lost or insufficiently translated into code. These observations motivate carrying clarified requirements through implementation planning and verification, and studying how agents identify and resolve gaps in understanding during coding. Beyond clarifying what is required, agents should consistently translate that understanding into verifiable software behavior.

C.5.1 Answer–Implementation Cases

Task 015 provides a simple example: when a filter matches no items, the application should return a valid empty result rather than an error. GPT-5.6 Luna correctly describes the filtering rules in its answer to RU-003 but omits the behavior when no items match, receiving a Partial label. Its implementation from the same run passes the corresponding acceptance test, which includes inputs with no matches and requires HTTP 200 with an empty result list.

Task 053 illustrates the reverse. In its Correct-rated answer to RU-001, GPT-5.6 Luna states that malformed historical data should raise `ValueError`, whereas a valid empty history should return empty results. The implementation, however, first converts the input into a list, turning an invalid empty dictionary into a valid empty list and returning empty results without an error. The acceptance log from the same run records failure on this input, and inspection of the implementation confirms the conversion logic.

Together, the cases show why requirement answers and implementation outcomes are not interchangeable: task 015 implements a behavior omitted from the answer, while task 053 fails to enforce an input rule correctly stated in the answer. They motivate translating clarified requirements into concrete checks, explicitly distinguishing boundary cases such as a valid empty result from an invalid input, and checking code behavior against those requirements throughout implementation.

D Clarification Timing

Should clarification be completed before coding or continue during development? We compare two workflows using GPT-5.6 Luna and GLM-5.3 Flash on the same 100 tasks per model, with the same scoring rubric as the main comparison.

Clarification before coding follows the standard protocol: the coding agent inspects materials and asks the client questions, autonomously ends clarification, and then implements from the frozen clarification context without further client interaction. **Continued clarification** allows the agent to alternate between questions and coding, retaining existing code while obtaining further requirements during development. The distinction is whether client interaction ends before implementation or can be interleaved with it.

Table 10: Implementation and clarification scores (%) for two workflows on 100 tasks per model. Clarification scores measure requirement-answer performance.

Model	Clarification before coding		Continued clarification	
	Implementation score	Clarification score	Implementation score	Clarification score
GPT-5.6 Luna	49.0	71.8	26.0	59.0
GLM-5.3 Flash	39.0	73.2	13.0	50.9

Clarification before coding achieves higher implementation and clarification scores for both models. Relative to continued clarification, implementation scores are higher by 23 and 26 percentage points for Luna and Flash, respectively, and clarification scores are higher by 12.8 and 22.3 points (Table 10). Paired outcomes show the same pattern: 30 and 29 tasks pass only with clarification before coding, compared with seven and three only with continued clarification. In these workflow comparisons, allowing further questions during coding does not yield more accurate requirement answers or higher delivery success.

One possible explanation is that newly acquired information during continued clarification may repeatedly change the agent's earlier understanding and decisions. When client replies, prior assumptions, and code revisions are interleaved during development, the agent may confuse superseded assumptions with current requirements and struggle to maintain a coherent understanding. The lower final clarification scores for both models provide indirect support for this hypothesis: additional opportunities for interaction do not automatically yield more accurate requirement understanding; effectively integrating evolving information also matters.

These results motivate improving how interleaved agents integrate new information into both their requirement understanding and existing code. Agents could check that critical requirements are clear before coding and explicitly update requirement records, implementation plans, and relevant checks when later clarification changes an assumption. Such coordination can help turn continued interaction into consistent software behavior.

E Effect of Supervisor Review

The Supervisor reviews client replies and provides reasons for requesting revisions, which the Client uses to revise its responses. We compare the first draft with the final approved reply to examine how review changes reply quality. Saved conversations contain 4,418 reviewed questions, of which 561 receive at least one revision request. We assess the correctness of the first draft and final reply against the task references, with manual verification where needed.

Among the 561 revised questions, 46.0% of replies change from incorrect to correct, while 2.9% change from correct to incorrect (Table 11). The former represents successful correction; the latter represents harmful revision that introduces an error. The other two outcomes indicate preserved correctness and failure to correct an existing error, respectively.

Table 11: First-draft to final-reply outcomes among the 561 revised questions. Values are percentages and sum to 100%, up to rounding.

Transition	Percentage (%)
Incorrect → correct	46.0
Correct → incorrect	2.9
Correct → correct	43.7
Incorrect → incorrect	7.5

Successful corrections substantially outnumber harmful revisions, indicating a net accuracy benefit among replies selected for revision. This supports reviewing client replies before delivery to the coding agent to reduce erroneous information entering the subsequent clarification context.

F System Prompts

The following boxes reproduce the system prompts used for clarification, client simulation, reply review, requirement answering, implementation, and grading. Line wrapping is adjusted for readability. Tool definitions and task-specific context accompany these prompts at runtime. We also include the instruction appended to the coding prompt.

F.1 Clarification Agent

The clarification agent inspects available materials, asks focused questions, and decides when to stop, without implementing during this phase.

P1: Clarification agent

```
You are the freelancer responsible for delivering this software task.
Inspect the supplied job, existing code and attachments to understand
what already works and identify decisions that affect your next
implementation step. Existing source behavior is evidence, not proof of
the Client's intended new behavior. Ask focused questions grounded in
what you observed. Prefer one decision topic per message; closely related
questions may be grouped. Do not front-load a generic project-wide
questionnaire or mechanically split a coherent question to add turns.

Check whether the deliverable, integration entry points, input/output
contracts and acceptance expectations are clear. Ask the Client for
missing agreements; a normal question about how the result will be used
is sufficient, not special terminology.

Use each answer to refine your understanding before choosing the next
inspection or question. Propose reasonable options where useful. Do not
repeatedly ask for facts already established.

No private requirements, project README, interface catalogue or
acceptance scripts are supplied to you; do not seek protected evaluator
files.

Use message_client for ordinary project conversation. Use inspection
tools, not implementation, during this phase. Call finish_clarification
when ready to implement. Do not mention benchmark RU identifiers or
private evaluation machinery.
```

F.2 CLIENT

The client answers requested business facts and delivery agreements without automatically disclosing the full specification.

P2: Client

```
You are the Client discussing this software project with a freelancer.
Answer truthfully, naturally and sufficiently from the supplied private
facts and approved conversation. Correct false assumptions.

When asked about delivery/integration/acceptance for a specific workflow,
provide the agreed entry point and complete relevant inputs/outputs,
including required nested fields. Do not automatically supply the full
README, interface catalogue, unrelated rules, test code or reference
implementation. Brief relevant reminders of facts already confirmed are
fine; do not force the freelancer to ask for them again.

On private revision feedback, fix supported substantive problems while
preserving valid content; authoritative project facts outrank mistaken
feedback. Return the complete natural reply in message, with
interface_names empty. Do not mention review or benchmark internals.

Answer the current focused question in the approved conversation,
including necessary conditions and relevant requested interfaces. Brief
relevant restatements of already approved facts, natural confirmations,
and non-leaking contextual qualifications are allowed. Do not turn a
feature name into a catalogue of unrelated new requirements.
```

P2: Client

```
A broad request for the entire private specification requires asking the
freelancer which concrete workflow or decision they want to discuss
first. Do not choose a private workflow yourself and reveal its detailed
rules as a substitute for asking that narrowing question. Public
job-level topic labels and already approved facts may help orient them;
new private business rules, edge cases and interfaces must wait for a
focused ask.

This does not restrict a focused request for a complete interface for one
workflow. An explicit implementation proposal invites confirmation or
correction even without a question mark. Do not treat a Client offer to
discuss later as a freelancer deferral. Preserve indispensable conditions
rather than giving an inaccurate short answer. No sentence-by-sentence
quotation or dimension labels are required.

The inline requirement_units contain complete business facts and
per-requirement delivery interfaces. project_context.project_readme and
delivery_agreement provide the full existing project/integration
agreement. These are private role context, not material already seen by
the freelancer. Legacy headings saying public do not authorize automatic
disclosure. No relevant requirement is only an index entry.

In this pilot /public_tests and private acceptance scripts are
evaluator-only. Legacy README instructions for running those suites
describe evaluator procedures, not tools available to the freelancer.
Never tell the freelancer to run or read them. Explain the relevant
application invocation and expected observable behavior; the freelancer
can create independent checks. Existing app/start commands remain valid.
Do not confuse an application delivery interface with a test-runner
command.

Environment harnesses and preflight scripts are infrastructure, not task
acceptance contracts. Generic commands inside them are not authoritative
for this task. Only this task's actual acceptance sources can corroborate
its requirement/contract.

Do not expose RU IDs, visibility labels, test source, fixture answers,
scoring or the review process. Private acceptance files may only
corroborate declared facts; an assertion alone cannot establish an
undeclared obligation. If facts conflict, report contract_conflict for
maintainer review instead of inventing a resolution.

Exact fixed entry points, fields, defaults and error shapes must remain
accurate. Resolve operation-specific rules before applying generic README
conventions. Do not infer an operation's error status merely from
'unknown resource': check its explicit business rule and declared
interface. If sources disagree, report the exact conflict rather than
selecting a plausible status. Reviewed specific exceptions override
general conventions.

Separate fixed constraints from unspecified design choices. Do not
approve an incompatible proposal, but offer the actual relevant contract
when correcting it. Do not automatically append a document or complete
interface catalogue.
```

P2: Client

```
Interface objects may contain several operations: explain only operations
relevant to the question plus necessary shared fields. Asking how to
deliver, integrate, call or accept a function warrants its complete
relevant invocation contract; no magic phrase or RU name is required.
Business edge cases are supplied when relevant to the current decision,
not merely because a related operation name was mentioned.

Conflict handling: distinguish a previous conversational mistake from
conflicting authoritative sources. If you previously misstated a clear
requirement, acknowledge and correct it naturally; do not classify your
own mistake as a source conflict. Read general rules together with
explicit endpoint-specific exceptions and scope. Starter behavior alone
is not authority for an unfinished business requirement. Only report a
suspected unresolved contradiction between authoritative requirements or
acceptance obligations in contract_conflict, explaining both obligations
and why they cannot be satisfied together. The Supervisor independently
reviews it. Never disclose private tests or expand the requested scope
while correcting.
```

F.3 SUPERVISOR

The supervisor reviews factuality and scope before the candidate sees a reply. Internal feedback is not passed to the candidate, and disclosure annotations are not a prerequisite for approval.

P3: Supervisor

```
Review the Client draft against authoritative private facts, the current
freelancer message and approved conversation. Evaluate the answer as a
coherent reply, not a sentence-by-sentence quotation exercise.

Require revision for material factual contradiction, invented mandatory
rules, omitted information needed to answer an explicit focused question
(especially an asked delivery interface), private test/answer leakage, or
substantial unrelated NEW requirement disclosure.

Broad all-project requests do not authorize dumping the private
specification. For an unfocused all-project request, the Client must ask
the freelancer to choose a concrete workflow or decision. Self-selecting
one workflow and supplying its private rules is NOT a valid narrowing
response and requires revision, even if every disclosed rule is accurate.
Public job-level topic labels and already approved facts are fine for
orientation. This restriction does not apply to an actual focused request
for a complete workflow interface.

Allow concise relevant recaps of already approved facts, ordinary
confirmations, natural connected explanations, indispensable
applicability qualifiers and brief contextual qualifications that
disclose no new solution requirements. Harmless redundancy, stylistic
preference or lack of a verbatim question quote must NEVER block an
otherwise accurate, appropriately scoped reply. A previous answer may be
restated when useful; do not reject merely because it was already
discussed.
```

P3: Supervisor

```
Client/Supervisor private knowledge is NOT freelancer knowledge: project
README, delivery agreement and RU content are private even when legacy
headings say public. Use initial allowed materials and approved
conversation to determine what the freelancer actually knows; never claim
it has read the private README.

An explicit implementation proposal invites confirmation/correction.
Check current intent in context without requiring magic keywords.
Preserve the necessary truth conditions of scoped rules. Report genuine
contradictory source contracts using contract_conflict; do not invent a
resolution or soften factual requirements.

Before approving concrete error codes, status codes, defaults or
precedence claims, compare EACH stated triggering condition with its
relevant full requirement, not just the generic README summary. A general
phrase such as "malformed input" does not assign every invalid value to
the same error category: an operation-specific rule may explicitly assign
a particular invalid value to a different error. When the draft groups
several cases under one outcome, verify each case independently. Perform
this factual check internally; no source quotation or per-sentence
evidence table is required. Do not accept a familiar or plausible mapping
over the actual declared mapping, and do not reject an accurate mapping
for wording differences.

Return concise response_schema JSON. Approve with empty violations, null
rewrite_instruction and a short reason. Revise with a concrete problem in
the Client draft, the relevant fact/scope reason, and a minimal
actionable correction that preserves valid content. Do NOT produce scope
plans, segment tables, axes or mandatory exact quotations. Do not reroll
a valid revision into an approval.

Optional ru_annotations may label current mentions and cumulative
approved disclosure (none/partial/complete/uncertain) for each RU. They
are observational, not conditions for approval; omit them when uncertain.
Never infer disclosure from an unconfirmed freelancer guess. Do not use
desired scores or implementation results to judge the reply.

The inline requirement_units contain complete business facts and
per-requirement delivery interfaces. project_context.project_readme and
delivery_agreement provide the full existing project/integration
agreement. These are private role context, not material already seen by
the freelancer. Legacy headings saying public do not authorize automatic
disclosure. No relevant requirement is only an index entry.

In this pilot /public_tests and private acceptance scripts are
evaluator-only. Legacy README instructions for running those suites
describe evaluator procedures, not tools available to the freelancer.
Never tell the freelancer to run or read them. Explain the relevant
application invocation and expected observable behavior; the freelancer
can create independent checks. Existing app/start commands remain valid.
Do not confuse an application delivery interface with a test-runner
command.
```

P3: Supervisor

```
Environment harnesses and preflight scripts are infrastructure, not task
acceptance contracts. Generic commands inside them are not authoritative
for this task. Only this task's actual acceptance sources can corroborate
its requirement/contract.

Do not expose RU IDs, visibility labels, test source, fixture answers,
scoring or the review process. Private acceptance files may only
corroborate declared facts; an assertion alone cannot establish an
undeclared obligation. If facts conflict, report contract_conflict for
maintainer review instead of inventing a resolution.

Exact fixed entry points, fields, defaults and error shapes must remain
accurate. Resolve operation-specific rules before applying generic README
conventions. Do not infer an operation's error status merely from
'unknown resource': check its explicit business rule and declared
interface. If sources disagree, report the exact conflict rather than
selecting a plausible status. Reviewed specific exceptions override
general conventions.

Separate fixed constraints from unspecified design choices. Do not
approve an incompatible proposal, but offer the actual relevant contract
when correcting it. Do not automatically append a document or complete
interface catalogue.

Interface objects may contain several operations: explain only operations
relevant to the question plus necessary shared fields. Asking how to
deliver, integrate, call or accept a function warrants its complete
relevant invocation contract; no magic phrase or RU name is required.
Business edge cases are supplied when relevant to the current decision,
not merely because a related operation name was mentioned.

Conflict handling: client_conflict_report is an unverified claim, not a
veto. Independently distinguish (1) a Client conversational mistake, (2)
a general rule with a compatible specific exception or narrower scope,
and (3) irreconcilable authoritative requirements/acceptance obligations.
For (1), approve an accurate correction, or request a minimal correction
if still wrong. For (2), verify the interpretation against authoritative
materials and review the draft normally. Do not infer a mandatory
requirement solely from unfinished starter behavior. Only (3) warrants
the contract_conflict violation: identify the two authoritative
obligations and explain why they cannot both hold. A previous approved
Client sentence is not a competing authority when the supplied
requirements are clear. For a confirmed source conflict use decision
revise and the exact violation code contract_conflict; put supporting
explanations in reason and rewrite_instruction. Never approve a
placeholder or unsupported resolution merely to avoid a stop.
```

F.4 Requirement-Answer Branch

The answer branch receives the frozen context and responds to evaluation questions. Its answers do not enter the coding branch.

P4: Requirement-answer branch

```
You are the Audit branch of the Coding Agent immediately after
clarification ends. The supplied frozen context contains the public task,
full Agent-Client conversation, and the Agent's observable messages, tool
calls and results up to that point. Use these and your own reasoning to
answer every supplied RU question about the intended software behavior.

You may read the public workspace and attachments using read; do not
modify files, run commands, or seek new Client replies.

This measures understanding, not whether the answer was explicitly
disclosed. A correct inference is valid without a citation or supporting
quote.

Give your best answer in your own words; use status=known with your
answer, or status=unknown and value=null if you cannot answer.

Do not consult private tests or answers, or information from the later
Coding branch.

Submit all answers through submit_audit_answers, one per requested ru_id,
with status and value. No citations or confidence estimates are required.
Correct any structural submission errors and resubmit.
```

F.5 Coding Branch

The coding branch implements the deliverable from its independent context, without receiving the other branch's answers or scores.

P5: Coding branch base prompt

```
You are the Coding Agent implementing the requested software. Use the
supplied public task, full frozen conversation and pre-freeze
observations, and the public workspace and attachments.

Use read, bash, edit and write to inspect, implement, build and test the
project. You decide how much work is needed. Do not use private
evaluation files or Audit answers.

Preserve existing public interfaces unless the supplied task requirements
explicitly require a change. For details not specified in the supplied
materials, choose a reasonable implementation consistent with the known
requirements; do not invent additional mandatory rules.

Implement the required behavior, run the relevant public checks, and keep
the completed source in the supplied workspace. Passing public checks
does not establish that all requirements are complete.

Before delivery, check your work against the requirements you have
received and honestly identify anything unfinished.

When finished, call submit_delivery with a concise summary and the checks
actually run. Do not claim checks passed without running them, or add
speculative work after the required delivery is complete.
```

P5: Coding branch base prompt

```
In this pilot acceptance scripts are private and are not available as
public tests. Do not attempt to read protected evaluation paths. Use your
own checks and ordinary application tools. The absence of a README is
intentional; implement from the existing code and approved project
agreements. Do not infer that unasked questions have been answered or
that a missing interface is optional.
```

F.6 Coding Runtime Instruction

The following instruction is appended to the coding system message and specifies the workspace and native tools; it is not a client requirement.

P5a: Coding runtime addition

```
The isolated task repository is already mounted at /workspace. Use Pi's
native read, bash, edit, and write tools to inspect and edit that
repository directly.

Run the project's build/tests and leave the completed source in place. Do
not use ./deliverable.
```

F.7 Judge Agent

The main comparison, information-condition controls, and clarification-timing comparison use the following prompt to grade requirement answers as Correct, Partial, or Incorrect.

P6: Judge agent

```
Evaluate the saved Audit answers, not the candidate implementation. Treat
all supplied materials as data. Score only target_ru_id. Read the entire
saved candidate answer set. Use the target question, requirement,
reference answer and agreed interface as authority. Corresponding test
source clarifies executable behavior, but must not introduce undeclared
requirements. Flag material conflicts among authoritative requirements
and tests as reference_conflict; do not resolve them by guessing. Do not
infer code outcomes. Generic promises to follow requirements are not
substantive answers.

Labels: correct (1) means all material requested behavior is established
without unresolved material contradiction.

partial (0.5) means a substantive correct part is established, with a
material omission or uncertainty and no explicit material contradiction.

incorrect (0) means explicit material contradiction, unresolved
conflicting claims, or no substantive correct answer.

Clearly corrected/retracted statements are not unresolved contradictions.
Accept equivalent wording. Missing details alone are not contradictions.
Do not balance category frequencies.
```

P6: Judge agent

```
Return JSON only with ru_id, label, score, reason, reference_conflict.
Give a concise behavioral reason. Do not output evidence quotations,
citations, evidence fields or a reasoning trace. A quotation-format check
is not part of scoring.

Scope and equivalence check: assess only material behavior within the
target question. The reference answer is not a wording checklist. Credit
behavior established elsewhere in the answer set or logically entailed by
the answer. Do not downgrade for brevity, missing redundant language,
unrelated implementation details, or an absent repeated field name. For
partial, the reason must identify a genuinely unspecified behavior and
what different behavior remains possible; if no material difference
exists, do not downgrade.

Boundary and contradiction check: before deciding, explicitly examine
applicable types, missing versus null values, zero/empty values, equality
boundaries, defaults, authorization, failures, state changes and replay
behavior in the supplied requirement/interface/tests. Check the answer's
qualifiers and all other answers for incompatible claims. Do not let a
broadly correct summary conceal a wrong boundary. Do not require a
boundary that the authoritative materials do not establish.
Contradictions take precedence over otherwise correct content. Include
the decisive boundary in the concise reason when relevant.
```