PRODBENCH: EVALUATING REQUIREMENT RECOVERY AND REALIZATION IN PRODUCER-ORIENTED SOFTWARE DEVELOPMENT

Anonymous authors
Paper under double-blind review

ABSTRACT

Real software requests leave crucial decisions unstated. We introduce PRODBENCH, a benchmark for evaluating an LLM *Producer* in agentic software development. A Producer turns an underspecified user request into an explicit implementation brief and oversees the development and validation of the resulting software. In our controlled protocol, the Producer gathers prior context, questions a fixed User Simulator, and directs a fixed Code Agent through implementation and revision. PRODBENCH contains 300 source-grounded Python tasks from 101 pseudonymized users, with 8,273 atomic requirements and 12,918 executable tests. Three matched request views vary the initial information while preserving each task's target. Five metrics measure requirement recovery, delivery completeness, preservation of recovered intent, targeted clarification, and revision gains. Two annotators audit construction quality and validate recovery judgments, with agreement between the panel and each annotator of 91.9% and 90.8%. Across 13 Producers, delivery completeness exceeds a request-only coder using the same backend by 19.9–38.7 percentage points on single-file tasks and 10.1–22.3 on repository tasks. Yet richer context improves recovery far more than delivery. These results identify a persistent gap between recovering requirements and realizing them in software.[1]

1 INTRODUCTION

Software can pass examples and still fail to meet a user's needs. A request to add an export function may leave output order, error handling, and compatibility unstated. Resolving these gaps requires recovering preferences, clarifying choices that belong to the user, and selecting routine defaults. Ambiguous requirements can lead generated code away from user intent, motivating targeted clarification (Mu et al., 2024; Rahmani et al., 2023). Studies of developer-agent collaboration further show the value of iterative interaction on complex tasks (Kumar et al., 2025). The challenge is to determine which requirements an agent should clarify with the user, which it should infer from context, and when it should stop gathering information or revising an implementation.

Existing benchmarks evaluate repository issue resolution, execution feedback, and dialogue with users (Jimenez et al., 2024; Yang et al., 2023; Yao et al., 2025). Recent work extends evaluation to underspecified requests, refinement, and user modeling (Vijayvargiya et al., 2026; King & Flanigan, 2026; Wang et al., 2026; Zhou et al., 2026b). Yet a final artifact score cannot distinguish a requirement that was never recovered from one that was communicated but lost during implementation. Intermediate evaluation helps locate such failures (Zhou et al., 2026a). To study the connection between recovery and realization, the requirements communicated before coding must also be linked to the behavior of the delivered software.

This distinction matters because the two cases have different causes and different remedies. An export-ordering requirement may be absent from the implementation brief, or it may be explicitly stated but remain unsatisfied by the generated code. These cases point to different potential bottlenecks in requirement acquisition, communication, and implementation. Conversely, a coder may satisfy an unstated requirement through its own conventions, so successful execution does not establish that

[1] Code is available at `https://anonymous.4open.science/r/ICLR_PRODBENCH-BE53`

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.

the requirement was recovered upstream. A single end-to-end score conflates all three situations: a pipeline that never asked the user, a pipeline that asked and mis-briefed, and a pipeline that briefed correctly but lost the requirement in code can be made to look identical. Evaluation therefore has to measure the brief and the artifact against the same fixed catalog, and record which evidence each requirement was expected to come from. Doing so requires controlled targets: each requirement needs a known recovery route, and each must be bound to tests that show whether it survived.

We introduce PRODBENCH to evaluate requirement recovery and realization in software development led by a *Producer*. A Producer is an agent responsible for turning an underspecified user request into an explicit implementation brief and overseeing the development and validation of the resulting software. Its responsibilities span **P**robing, Requirement **R**ecovery, **O**rchestration, and **D**elivery. In our protocol, the Producer retrieves prior context, asks a fixed User Simulator for clarification, briefs a fixed Code Agent, and reviews its outputs. Holding these services fixed supports comparison of Producers under a shared implementation configuration (Zhu et al., 2025). Because the brief is logged before coding begins and the artifact is scored afterward against the same frozen catalog, the protocol attributes a missing requirement to acquisition or communication rather than to the coder, and attributes a realized-but-unstated requirement to the coder's own conventions rather than to successful recovery. Repeating each task under three matched views with different initial information then shows whether recovery responds to the information available.

PRODBENCH comprises 300 Python tasks from 101 pseudonymized users, spanning 200 single-file and 100 repository tasks. Model-assisted construction converts source interactions and admissible historical evidence into frozen targets with 8,273 atomic requirements and 12,918 executable tests. Each hidden requirement has an intended recovery source in prior context, user clarification, or an engineering default. These are constructed evaluation targets, with source grounding and recoverability examined by human audit. Three matched request views provide a sparse request at L0, unresolved dimensions at L1, and qualitative guidance at L2 while keeping requirements and tests fixed.

Five complementary metrics follow the process from information gathering to delivery. *Intent Recovery* measures requirements communicated in the brief. *Delivery Completeness* measures realized requirements, while *Recovered Intent Realization* measures preservation of recovered intent in the artifact. *Interaction F1* assesses targeted clarification, and *ReviewGain* measures changes in delivery through revision. Reporting them separately distinguishes broad recovery from faithful implementation of a narrow subset.

Our contributions are summarized below.

- **A benchmark linking requirement sources to executable behavior.** We release 300 tasks with atomic requirements, annotated recovery sources, matched request views, and requirement-bound tests. This structure supports comparisons of information use against a fixed target and connects the content of a brief to the resulting software.
- **A controlled and audited Producer evaluation protocol.** Shared user and coding services separate Producer decisions from changes in the implementation backend. Five metrics assess recovery, realization, clarification, and revision. Two annotators audit 90 construction packets and validate recovery judgments on 360 requirement–brief pairs, with panel agreement of 91.9% and 90.8%.
- **Evidence of a persistent gap between recovery and realization.** Across 13 Producers and 11,700 episodes, every Producer exceeds the request-only coder's mean delivery completeness on both tracks. Richer requests improve recovery more than delivery, and positive mean revision gains coexist with regressions. These findings identify preservation of recovered intent and reliable review as priorities beyond gathering more information.

2 RELATED WORK

Software and interactive coding benchmarks. SWE-bench evaluates repository issue resolution, while SWE-agent, OpenHands, and Agentless study different implementation workflows (Jimenez et al., 2024; Yang et al., 2024; Wang et al., 2025b; Xia et al., 2025). SWE-Gym and SWE-smith support agent training, and SWT-Bench and LiveCodeBench evaluate testing and coding capabilities (Pan et al., 2025b; Yang et al., 2025; Mündler et al., 2024; Jain et al., 2025). These settings establish

Table 1: **Benchmark scope.** ✓: direct. ▲: partial. ×: unsupported. Routing annotates recoverable requirement sources. Checks explicitly link requirements to executable tests. Stagewise scores assess intermediate production capabilities. Fixed-coder comparisons share the same downstream implementer.

	Task Setting		Intent Evidence		Evaluation Design			
Benchmark	**Underspecified**	**Dialogue**	**History**	**Source-Routed**	**Requirement-Linked Checks**	**Stagewise Evaluation**	**Interaction-Cost Metric**	**Fixed-Coder Comparison**
SWE-bench	×	×	×	×	×	×	×	×
Ambig-SWE	✓	✓	×	×	×	✓	✓	×
Dialogue SWE-Bench	✓	✓	×	×	×	▲	✓	×
Asuka-Bench	✓	▲	×	×	✓	▲	×	×
ICAE-Bench	✓	✓	×	×	▲	✓	✓	×
SWE-RPG	✓	×	×	×	▲	✓	×	×
Stateful SWE	✓	✓	✓	×	×	▲	×	▲
LongMemEval	×	×	✓	×	×	×	×	×
τ-bench	▲	✓	×	×	×	×	×	×
PRODBENCH	✓	✓	✓	✓	✓	✓	✓	✓

executable evaluation but do not by themselves distinguish missing requirements from failures to implement known ones. ClarifyGPT, Ambig-SWE, When Benchmarks Talk, and Dialogue SWE-Bench examine clarification and dialogue (Mu et al., 2024; Vijayvargiya et al., 2026; Pan et al., 2025a; King & Flanigan, 2026). Asuka-Bench and ICAE-Bench extend evaluation to refinement and interactive project building (Wang et al., 2026; Peng et al., 2026). Dialogue quality, functional checks, and interaction metrics reveal complementary capabilities. However, when the evaluated agent also implements the software, delivery differences reflect both information gathering and coding. Moreover, successful dialogue does not establish that the communicated requirements survive implementation. PRODBENCH addresses this measurement gap by comparing Producers under a shared Code Agent and scoring both the brief and the realization of its recovered requirements.

Intent recovery, memory, and evaluation. SWE-RPG provides requirement and planning references for diagnosing intermediate failures (Zhou et al., 2026a). ToM-SWE separates user modeling from coding and introduces Stateful SWE with prior interaction histories (Zhou et al., 2026b). These approaches motivate our focus on intermediate decisions and contextual evidence. Stage diagnosis alone, however, does not measure whether each requirement should be recovered from history, clarified with the user, or supplied as a default. Likewise, access to history does not establish whether the recovered preferences appear in the delivered artifact. Conversational memory benchmarks and systems examine retention and retrieval (Maharana et al., 2024; Wu et al., 2025; Tan et al., 2025; Xu et al., 2025; 2026), while tool interaction and iterative refinement study complementary parts of the workflow (Yao et al., 2025; Madaan et al., 2023; Chen et al., 2024; Shinn et al., 2023). PRODBENCH connects source labels, brief recovery, and executable realization while retaining interaction costs and revision outcomes. Table 1 summarizes this scope. Following work on benchmark validity and automated judging (Zhu et al., 2025; Li et al., 2025; Wang et al., 2025a), we validate construction and recovery judgments separately. Majority voting reduces dependence on one judge, while human agreement provides an external check.

3 PRODBENCH

Evaluation objective. PRODBENCH evaluates how a Producer gathers evidence, prepares an implementation brief, and guides a fixed Code Agent through delivery. Five metrics score these decisions. Each task has one frozen target and three matched request views, so changes in initial information can be compared against the same requirements and tests.

3.1 SOURCE-GROUNDED TASK CONSTRUCTION AND QUALITY CONTROL

Source evidence and task grounding. The 300 tasks originate from pseudonymized software-development interactions. Each supplies a task kernel, an incomplete programming intent converted into a self-contained Python task. Profile and history evidence must precede the source interaction, exclude it and related sessions, and retain source identifiers. History-based preferences typically require support from at least two sessions. Conversion removes identifying context and supplies a stable callable contract. The result is a constructed target grounded in source evidence, with model-assisted engineering choices.

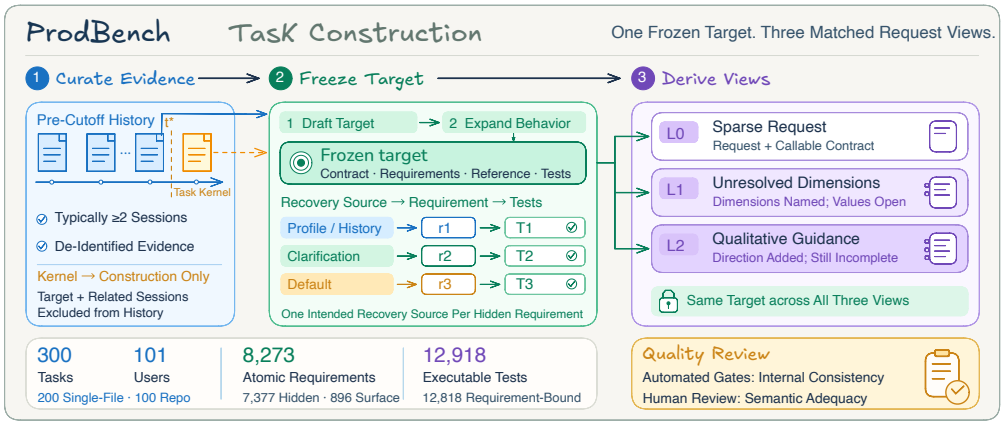


Figure 1: **Task construction overview.** Source-grounded targets freeze contracts, atomic requirements, reference code, and tests before deriving L0–L2 views. Automated gates check consistency. Human review assesses semantic adequacy.

Target construction and validation. `DeepSeek-V4-Pro` first drafts a specification, contract, reference implementation, tests, and core atomic requirements. A second pass identifies further behavioral decisions. Automated gates check contract and test consistency, reference execution, determinism, purity, naive-solution discrimination, temporal separation, de-identification, leakage, and track structure. Repairs are logged, and human review separately assesses semantic faithfulness and recoverability in Section 4.1. Task i has frozen target $y_i = (K_i, E_i, R_i, S_i, T_i)$, comprising contract, admissible evidence, requirements, reference implementation, and tests. All are fixed before view derivation. Here, *gold* denotes these construction-derived references, not requirements individually confirmed by the source users. They combine historical evidence and model-drafted choices. Appendix B.1 distinguishes their provenance from the human audit and entailment labels.

Atomic requirements and recovery sources. Surface requirements specify mechanical constraints such as signatures and output shapes. Hidden requirements are behavioral choices left open in L0. Each has an intended recovery source in clarification, profile/history, or default. For an export request, history may support an ordering preference, clarification may resolve a destination choice, and a default may determine empty-input behavior. Source labels and test bindings connect information use to executable realization, following requirements traceability (Gotel & Finkelstein, 1994). The labels specify evaluation routes rather than uniquely valid real-world choices. Defaults fix one convention, and targeted questions can still elicit answers about them.

Matched views and executable evaluation. Views $x_i^{(0)}, x_i^{(1)}, x_i^{(2)}$ share the frozen target. L0 preserves the sparse request and contract. L1 names unresolved dimensions, and L2 adds qualitative direction while withholding literal test-checked answers. The human audit examines this separation and records exceptions. Only clarification and default targets receive hints. History targets remain tied to prior context. Cross-view checks reconcile all requirement bindings. For artifact A, $g_i(r, A) = 1$ only if all tests bound to requirement r pass. Failures, skips, unrun cases, timeouts, and import errors yield zero. Each requirement has equal weight.

Construction example. Task `s3ts_020` begins from this one-sentence request: *"I need a function that will take scraped product info organized by category buckets and a config, and return cleaned products, a step log, and a temp files placeholder."* It names three outputs and no behavior, yet the frozen target for `process_products` carries 22 atomic requirements bound to 40 tests: 1 surface and 21 hidden (8 clarification, 11 default, 2 profile/history), each routed to evidence that can settle it. Brand normalization is routed to profile/history: the history records that the user "discovered five published articles spelled the brand as 'AISunny'; they corrected it to 'Aisunny'," so `r5` freezes case-variant replacement to `Aisunny` with no hint at any view. Blacklist handling is routed to clarification: `r3` freezes case-insensitive removal with normalized whitespace, named but not resolved at L1/L2, so `test_r3_0` grades the exact string `'', great product''`. A default sets `rx20_6` to an empty string for an absent `title`, reachable only through hints. Appendix B.7 traces this task end to end.

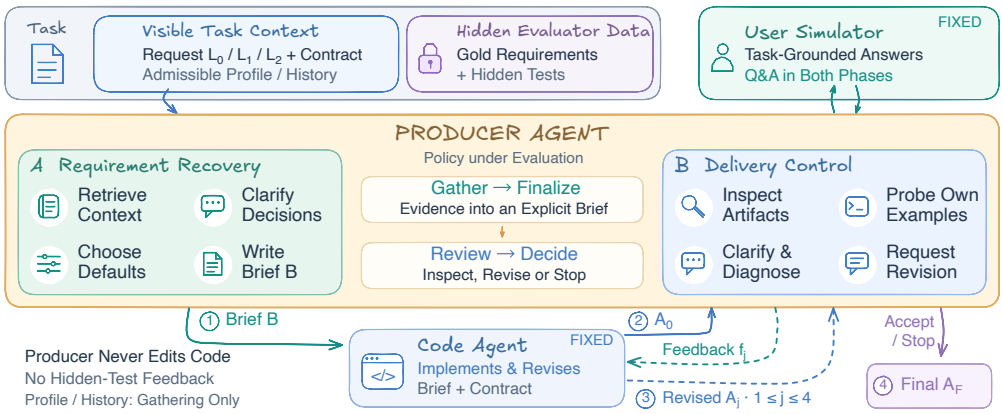


Figure 3: **Producer evaluation workflow.** The Producer queries the User Simulator, briefs the Code Agent, and reviews outputs. The simulator accesses hidden task information. The Producer and coder receive neither the construction-generated requirement catalog nor hidden-test feedback.

3.2 Benchmark Inventory and Task Mix

Tasks and requirement inventory. The benchmark contains 300 tasks from 101 pseudonymized users, with 200 single-file and 100 repository tasks. Its 8,273 atomic requirements comprise 896 surface and 7,377 hidden. Hidden targets include 3,948 clarification (53.5%), 640 profile/history (8.7%), and 2,789 default (37.8%), with at least one clarification and one profile/history target per task. These proportions describe the constructed mix rather than real-world source prevalence.

Task families and test coverage. Figure 2 shows the task-family distribution. The 12,918 tests comprise 12,818 requirement-bound cases and 100 repository package checks retained only as diagnostics. Single-file tasks have 15–28 hidden requirements and 20–49 tests. Repository tasks have 24–75 and 40–134, respectively. Since tracks contain different tasks, their performance differences cannot be attributed solely to file count (Zan et al., 2025; Badertdinov et al., 2025). Appendix B.5 reports the full inventory.



Figure 2: **Task-family mix.**

3.3 Producer Evaluation Workflow

Shared roles and user simulation. Each episode evaluates one Producer on task i at view $k \in \{0, 1, 2\}$ under fixed prompts, interfaces, budgets, and role services. Figure 3 shows the workflow. The Evaluator scores the episode without supplying feedback to the Producer or Code Agent, and no Producer-facing role observes the reference catalog or hidden-test verdicts. The simulator matches a question to at most one hidden requirement. Clarification matches receive answers based on construction-generated references, not live user feedback. History matches redirect the Producer to prior context. Default and unmatched questions receive specification-grounded answers restricted to the question. Multi-part questions can earn credit for only one target.

Evidence gathering and briefing. The Producer starts with $x_i^{(k)}$ and K_i. It retrieves admissible context through `read_profile`, `search_history`, and `read_history`, and queries the simulator through `ask_user`. Gathering ends with a free-form brief B_i submitted through `finalize`. The brief enters both recovery judging and implementation. Defaults appear in the brief or subsequent feedback. If no usable brief is submitted, the runtime can use the request and gathered dialogue as a fallback. These episodes retain artifact scores, with explicit finalization recorded separately.

Implementation, review, and revision. The Code Agent implements B_i under K_i and returns A_0. The Producer can inspect code, run self-authored examples, ask follow-up questions, request revision, or accept. It cannot edit files, retrieve more history during review, or observe hidden-test verdicts. Questions share a quota across both phases. Each of at most four revisions supplies feedback f_j, the

original brief, and A_{j-1} to the coder. A usable replacement becomes A_j. Otherwise $A_j = A_{j-1}$. The current artifact is delivered at termination even without explicit acceptance. Logged questions, feedback, artifacts, and stopping decisions support analysis of repair and regression without assuming that refinement is reliably corrective (Huang et al., 2024).

3.4 Evaluation metrics

Five measures follow one episode from information gathering to delivery. *Intent Recovery* (IR) asks how much of the requirement catalog the Producer expressed in its brief; *Delivery Completeness* (DC), how much of that catalog the delivered artifact realizes end to end; *Recovered Intent Realization* (RIR), how much of the recovered intent survived implementation; *Interaction F1* (INT-F1), whether questions targeted the open clarification requirements; and *ReviewGain* (RG), whether review improved the artifact over the initial implementation. Two shared terms underlie them. Write R_i for the atomic requirements of task i and $R_i^{\mathrm{hid}} \subset R_i$ for its hidden requirements. Each judge independently decides whether requirement r is entailed by its statement and the brief B_i, without inspecting code or test outcomes, following established LLM-as-a-judge practice (Li et al., 2025; Wang et al., 2025a; Zhuge et al., 2025). GPT-5.5, Gemini-3.5-Flash, and DeepSeek-V4-Flash cast binary votes $v_{ij}(r, B_i)$, and a requirement is *expressed* when at least two are entailed, giving recovered set $\widehat{R}_i = \{r \in R_i^{\mathrm{hid}} : \sum_j v_{ij}(r, B_i) \geq 2\}$. It is *realized* when the artifact passes every test bound to it: for artifact A, $g_i(r, A) = 1$ only if all tests bound to r pass. Failed, skipped, or unrun cases yield zero, with equal weight per requirement. Because the recovered set is fixed before coding, requirements first raised during review never enlarge it.

Intent Recovery (IR). $\mathrm{IR}_i = |\widehat{R}_i| \,/\, |R_i^{\mathrm{hid}}|$: expressed hidden requirements over all hidden requirements, insensitive to surface requirements and to the coder.

Delivery Completeness (DC). $\mathrm{DC}_i(A) = \sum_{r \in R_i} g_i(r, A) \,/\, |R_i|$: realized requirements over all atomic requirements, including surface ones, so DC credits behavior the coder supplies on its own.

Recovered Intent Realization (RIR). $\mathrm{RIR}_i(A) = \sum_{r \in \widehat{R}_i} g_i(r, A) \,/\, |\widehat{R}_i|$. Both terms range over the recovered set, so RIR scores preservation of recovered intent under the shared coder, reflecting briefing, implementation, and review together. If $\widehat{R}_i = \varnothing$, RIR is undefined and the episode is omitted from that metric's aggregation rather than scored zero; edge cases and eligibility counts are in Appendix E.

Interaction F1 (INT-F1). $\text{INT-F1}_i = 2h_i/(q_i + c_i)$, where h_i is the number of distinct validated clarification targets, q_i the submitted questions in both phases, and $c_i > 0$ the fixed clarification-target count. Repeated, profile- or default-routed, and unmatched questions raise q_i but not h_i, and asking none yields zero, so the score credits targeted clarification only.

ReviewGain (RG). $\mathrm{RG}_i = \mathrm{DC}_i(A_F) - \mathrm{DC}_i(A_0)$: the change in delivery from the initial artifact to the delivered one, including zero and negative changes. Episodes without revision contribute zero by construction, so RG measures the net contribution of review rather than its frequency. The five measures are reported separately, and judging and hidden execution never enter the Producer's observations or feedback.

4 Experiments

We study construction and recovery-judge reliability in RQ1, recovery and delivery in RQ2, recovery channels and interaction burden in RQ3, and questioning and revision trajectories in RQ4.

Shared protocol and statistics. We evaluate the 13 Producers in Table 2 over all $13 \times 300 \times 3 = 11{,}700$ episodes, each with an integrity-verified brief and complete three-judge IR scoring. The Code Agent is `Kimi-K2.7-Code` and the User Simulator `DeepSeek-V4-Flash`; budgets allow 40 actions, including 18 gathering actions, 14 questions, 6 example executions, and 4 revision rounds. For each track we average the three view scores within each task and then across tasks, with pooled summaries weighting single-file and repository tracks 2:1. Intervals use 10,000 paired, track-stratified task-bootstrap draws with a source-user-clustered sensitivity analysis (Efron, 1992); Tables report

point estimates and intervals appear where shown. Appendix G gives metric intervals and paired model contrasts, and Appendix E details eligibility rules, subset averaging, and level contrasts.

4.1 RQ1. Construction quality and recovery-judge validation

Construction quality. Two annotators reviewed all 90 sampled construction packets, covering 60 single-file and 30 repository tasks to support a diagnostic audit of construction quality. Table 3A reports each annotator's mean compliance across eight checks. Binary checks are summarized as task pass rates, while fraction-valued checks are averaged within tasks and then across the sampled packets. Both annotators report mean compliance of at least 94.4% on every check. Temporal separation and reference validity pass on all 90 packets for both annotators. Appendix D.3 provides the sampling design and detailed audit protocol.

Table 3: **Human validation.** A: mean within-task construction compliance (%; 90 packets). B: recovery-panel metrics against each annotator (%; 360 pairs).

A. Human construction audit (90 packets)		
Check	Annot. A	Annot. B
Source grounding	100.0	94.4
Temporal separation	100.0	100.0
Visibility policy	99.2	99.0
Channel assignment	96.6	95.8
L1 non-directionality	100.0	99.8
L2 usefulness	100.0	99.8
L2 non-literalness	97.9	95.0
Reference validity	100.0	100.0
B. Recovery-judge validation (360 pairs)		
Panel metric (%)	Annot. A	Annot. B
Agreement	91.9	90.8
Precision	95.8	91.0
Recall	91.0	93.2
F1	93.3	92.1

Recovery-judge validation. Two annotators independently assess 360 hidden-requirement–brief pairs, blinded to Producer identities, panel votes, and each other's labels. Sampling spans track, Producer, view, and original unanimous-label strata. Table 3B reports majority-panel agreement of 91.9% and 90.8% against annotators A and B, respectively, with corresponding F1 scores of 93.3% and 92.1%. Agreement against A is 92.0% under the original sampling weights. Agreement between annotators is 95.0%, with Krippendorff's $\alpha = 0.896$ and Cohen's $\kappa = 0.896$ (Krippendorff, 1989; Cohen, 1960). IR rankings remain stable when any judge is omitted, with Spearman $\rho \geq 0.96$. Appendix D reports the sampling and weighting protocol, aggregation-rule comparison, and panel-composition analyses.

4.2 RQ2. Recovery, information preservation, and delivery

Primary comparison. Table 2 compares all Producers with a request-only reference. The same `Kimi-K2.7-Code` backend receives each view verbatim in one implementation call with output limits of 16k and 24k tokens for the two tracks, without questioning, probes, or revision. Every Producer improves DC over this reference: single-file DC rises from 24.9% to 44.8–63.6% (+19.9 to +38.7 pp), while repository DC rises from 28.4% to 38.5–50.7% (+10.1 to +22.3 pp). This consistent improvement across all thirteen Producers establishes the value of the Producer workflow under a shared coding backend. The reference's per-view DC is 20.5/24.8/29.4% on single-file tasks and 26.7/29.3/29.3% on repositories (Appendix H). DC captures end-to-end delivery, while IR, RIR, INT-F1, and ReviewGain decompose Producer performance into requirement recovery, preservation through implementation, targeted clarification, and revision.

The per-model ordering is less informative than the three patterns it conceals. First, recovery and delivery are only loosely coupled: 27.9% of single-file DC and 52.5% of repository DC comes from requirements outside the recovered set, either unstated hidden requirements the coder supplied through its own conventions or contract surface requirements. That share falls as IR rises ($\rho = -0.96$ single-file, $\rho = -0.97$ repository), and Gemini-3.5-Flash is the limiting case: the lowest single-file IR (31.4%) with DC reaching 48.5%, 51.1% of it from outside its recovered set. Low IR therefore does not imply a weak artifact. Second, ReviewGain is not a clean measure of review capability, because it bundles room to improve with the effectiveness of edits: models with lower initial DC gain more ($\rho = -0.75$ repository), and Gemini-3.5-Flash's leading 19.5 pp accompanies the second-lowest initial DC (29.0%). At a matched single revision, gain per step spans only 10.7–13.4 pp. Third, equal final DC can arise from opposite processes. DeepSeek-V4-Flash and Gemini-3.5-Flash finish single-file within 0.3 pp (48.2% and 48.5%), yet recover 50.1% versus 31.4% of hidden requirements across 7.3 versus 3.5 questions; on repository tasks Gemini-3.5-Flash and Kimi-K3 finish within 0.6 pp despite a 28.8 pp IR gap. Delivery completeness alone cannot distinguish these paths.

Table 2: **Five-metric comparison by task track.** IR, DC, RIR, and INT-F1 in %; RG (ReviewGain) in pp. Reference row on top (n/a = inapplicable); Producers below, best **bold**, second underlined.

Producer / ref.	Single-file (200)					Repo (100)				
	IR	DC	RIR	INT-F1	RG	IR	DC	RIR	INT-F1	RG
Request-only coder ref.	n/a	24.9	n/a	n/a	n/a	n/a	28.4	n/a	n/a	n/a
Claude-Fable-5	**72.1**	62.3	72.4	**39.5**	4.9	**64.5**	**50.7**	51.3	**28.7**	8.4
Claude-Opus-4.8	61.0	52.5	63.4	34.5	5.6	56.6	46.9	48.5	25.1	8.3
Gemini-3.8-Flash	65.2	**63.6**	**76.8**	29.2	5.9	54.3	50.4	**54.0**	21.7	8.9
Gemini-3.5-Flash	31.4	48.5	58.8	20.8	**19.5**	28.3	44.4	44.5	13.9	**15.2**
GPT-5.6-Terra	68.0	57.7	67.7	32.3	11.4	62.3	46.1	47.2	24.5	11.8
GPT-5.5	69.2	59.2	69.7	33.4	6.2	63.1	47.5	48.5	25.0	11.4
DeepSeek-V4.1-Flash	69.4	63.3	73.7	34.6	8.4	62.7	48.7	50.2	27.3	9.0
DeepSeek-V4-Flash	50.1	48.2	61.4	32.0	9.4	47.1	43.3	45.1	23.9	9.5
DeepSeek-V4-Pro	60.4	52.8	63.6	28.8	5.4	58.2	46.3	46.2	23.9	9.0
Qwen3.5-Plus	57.9	55.0	67.2	33.1	6.0	47.6	46.0	49.3	23.3	8.9
Kimi-K3	61.2	51.9	62.8	31.8	6.0	57.1	45.0	45.4	23.5	10.3
GLM-5.2	54.8	52.3	66.3	29.5	6.8	51.9	46.4	48.3	21.5	7.5
MiniMax-M3	58.6	44.8	57.5	26.5	4.9	53.5	38.5	37.9	20.5	9.9

Paired information contrasts. Figure 4 spans the strongest and weakest level responses, with the full grid in the release. Richer views act mainly on recovery: IR rises $+8.1$ to $+24.6$ pp at L1 and $+29.1$ to $+43.6$ pp at L2, while DC changes range from -1.1 to $+6.1$ pp at L1 and $+5.8$ to $+17.3$ pp at L2, and RIR moves between -8.7 and $+5.8$ pp. These recovery gains identify implementation preservation as the central opportunity: making requirements available is only the first step toward delivering them. High RIR with low IR highlights reliable implementation of a narrow subset, whereas high DC with lower IR can reflect coder-supplied behavior beyond the brief. Gemini-3.5-Flash has the lowest IR at 31.4/28.3% yet delivers DC within a few points of the median.

4.3 RQ3. Recovery channels and user-facing interaction

Channel realization and acquisition traces. At L0, channel realization averages $g_i(r, A_F)$ over hidden requirements assigned to each channel and then over supporting tasks. The route-consistent diagnostic further requires a validated matched question for clarification targets and no match for profile/history or default targets. In Figure 5, profile/history targets achieve the highest realization for twelve of thirteen models, reaching 59.4% for DeepSeek-V4.1-Flash, while defaults rank lowest for nine. For clarification targets, route-consistent realization exceeds the channel mean by 11–20 pp for every model, including 62.2% versus 48.1% for Claude-Fable-5, associating validated questions

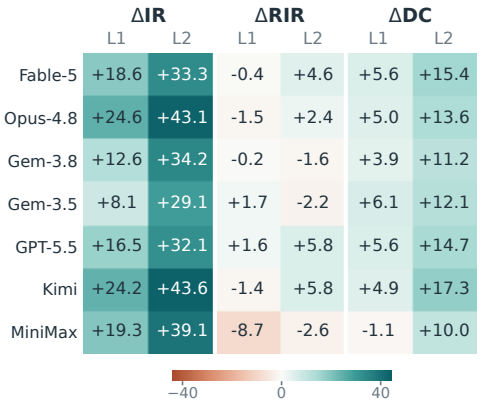


Figure 4: **Paired changes from L0** (pp). Rows: seven Producers. Columns: L1/L2 contrasts for IR, RIR, and DC. Both tracks pooled. RIR uses eligible pairs.

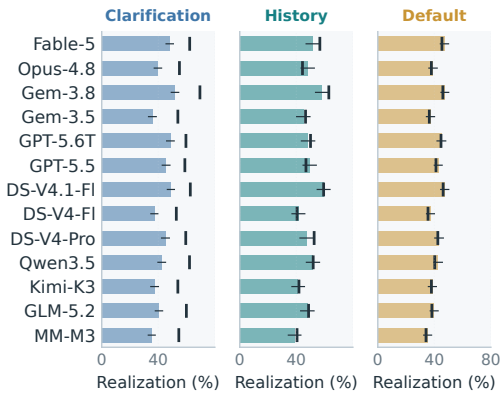


Figure 5: **Final realization by source channel at L0.** Bars: mean realization with 95% task-bootstrap intervals. Markers: route-consistent subsets.

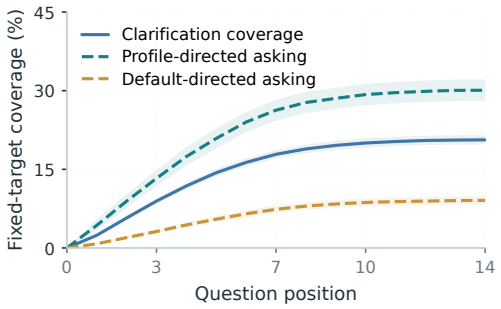


Figure 6: **Coverage over question position at L0.** 3,900 episodes, 13 models. 95% task-bootstrap intervals.

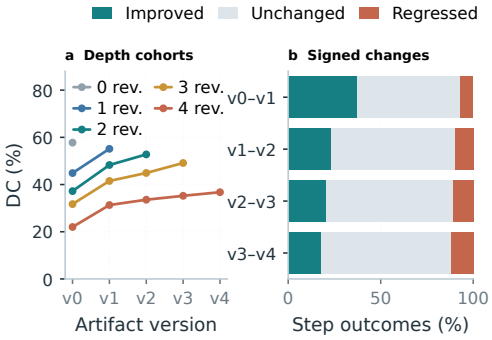


Figure 7: **Delivery through revision.** (a) DC by version within final-depth cohorts. (b) Signed step outcomes.

with substantially higher downstream realization. Together, these traces motivate channel-aware acquisition that pairs contextual evidence with targeted clarification.

Burden and yield. Questions per episode range from 3.2 to 9.1 and pooled hit yield from 0.30 to 0.39 per question. Gemini-3.8-Flash achieves 2.50 hits per 1,000 estimated dialogue tokens, versus 1.06 for DeepSeek-V4.1-Flash, whose mean dialogue is 2,772 tokens per episode against 931. Off-route, repeated, and unmatched questions account for 17.0–23.6%, 9.1–20.1%, and 26.3–36.2% of submissions. Per-model question volume, dialogue cost, and hit yield appear in Table A12.

4.4 RQ4. Question horizons and coder-facing review

Question trajectories. Figure 6 normalizes matched targets by each task's channel support. Of 3,900 L0 episodes, 3,896 ask at least one question, yet only 149 use the full budget of 14, and final coverage is 30.1% for profile-directed asking, 20.6% for clarification, and 9.1% for defaults. Questioning is nearly universal but sustained questioning is rare, allocation is uneven across channels, and all curves flatten after roughly seven questions—a pattern of target saturation that motivates questioning adapted to remaining information needs. Appendix J reports the per-channel curves.

Review gains and regressions. Mean DC improves within every nonzero-depth cohort, and ReviewGain is positive for all thirteen models, ranging from $+4.9$ to $+19.5$ pp on single-file tasks. Producers concentrate deeper revision on artifacts with greater room to improve: initial DC is 57.8% for zero-revision episodes and 22.0% for four-revision episodes. The yield of additional edits declines, with improving steps falling from 37.4% to 18.1% and regressions rising from 6.9% to 12.0%. This motivates review policies that prioritize unresolved requirements while preserving already-correct behavior, and stopping rules sensitive to that declining yield. Depth cohorts describe policy-selected trajectories rather than causal effects of extra revisions (Huang et al., 2024; Zhou et al., 2026b; Chen et al., 2024). Appendix K gives the cohort gains and version-selection accounting.

5 Conclusion

Agentic software development requires deciding what to probe, which requirements to recover, how to brief the coder, and when to revise or stop. ProdBench evaluates these decisions with 300 source-grounded tasks, 8,273 atomic requirements, three paired request views, and five complementary metrics, supported by construction audits and recovery-judge validation. Across thirteen Producers, all achieve higher mean delivery completeness than a request-only coder on both tracks. Richer views improve recovery more than delivery, while revision yields positive average gains despite individual regressions. The shared user and coder backends make Producer strategies directly comparable against fixed targets, revealing distinct strengths in acquisition, delivery, and review. The persistent gap between recovered and realized requirements highlights the need to improve both intent recovery and its preservation through implementation. Released records support offline verification, while the workflow enables fresh evaluation of how Producers gather information and manage revision.

AI USE STATEMENT

Generative AI tools were used in three roles. For *benchmark construction*, task specifications, contracts, reference implementations, tests, and atomic requirements were drafted by `DeepSeek-V4-Pro` in the two-pass procedure of Section 3.1, then screened by automated acceptance gates, with 90 sampled packets audited by two independent human annotators as described in Section 4.1. For *evaluation infrastructure*, fixed LLM services act as the User Simulator, the Code Agent, and the three recovery judges as described in Section 3.3. They are part of the measured instrument, and their behavior is examined in Sections 4.1 and 4.3. For *analysis and writing*, AI tools assisted with data analysis and with drafting and polishing the text. A GPT-5.5-simulated pilot of the judge-validation workflow served only as a workflow diagnostic, not as validation evidence. Reported numbers come from the evaluation pipeline, not manual editing. We have reviewed all AI-assisted content and take responsibility for the final content of this work.

ETHICS STATEMENT

PRODBENCH draws on software-development conversations from users who opted in to having their feedback and interaction data used for model improvement under the model service's user agreement. Source data collection was restricted to these consenting users. To protect privacy, the release excludes raw source conversations, source-session metadata, and identifying traces; task-local user context is de-identified and user identities are replaced with pseudonyms. Construction checks include de-identification, with the audit protocol described in Appendix D.3. Released histories retain the context needed for the benchmark, and contact literals use placeholders or reserved-domain fixtures. Reference requirements and simulator replies are constructed benchmark targets and model outputs, rather than additional statements elicited from the original users. The benchmark is intended to study requirement recovery and software realization; released context should not be used to identify or profile source users. Appendix O and the accompanying data card document the release contents and data boundaries.

REPRODUCIBILITY STATEMENT

The accompanying anonymous code package includes all 300 task packages, construction and evaluation workflows, prompts, configurations, frozen records for 11,700 episodes, requirement-level judge votes and execution outcomes, and anonymous human-validation labels. Section 4 specifies the shared protocol and statistical procedure; Appendix O and `docs/REPRODUCIBILITY.md` map the reported results to their evidence and reproduction commands. After dependency installation, `bash reproduce.sh` recomputes the main metrics and regenerates the main exhibits without model calls; `bash reproduce_appendix.sh` extends these checks to the appendix and prompts. Reference validators and the released workflow support executable task validation and fresh evaluation with configured model endpoints. Offline reproduction verifies recorded measurements; fresh evaluation generates new artifacts and judgments. Raw source conversations and complete historical model artifacts are excluded, as detailed in Appendix O.

REFERENCES

Ibragim Badertdinov, Alexander Golubev, Maksim Nekrashevich, Anton Shevtsov, Simon Karasik, Andrei Andriushchenko, Maria Trofimova, Daria Litvintseva, and Boris Yangel. Swe-rebench: An automated pipeline for task collection and decontaminated evaluation of software engineering agents. In *Advances in Neural Information Processing Systems*, volume 38, Main Conference. Curran Associates, Inc., 2025. doi: 10.52202/085713-0788. URL `https://proceedings.neurips.cc/paper_files/paper/2025/file/21bec6ace947b1b58967b945c8ac0f10-Paper-Datasets_and_Benchmarks_Track.pdf`.

Xinyun Chen, Maxwell Lin, Nathanael Schaerli, and Denny Zhou. Teaching large language models to self-debug. In *International Conference on Learning Representations*, volume 2024, pp. 8746–8825, 2024. URL `https://proceedings.iclr.cc/paper_files/paper/2024/file/2460396f2d0d421885997dd1612ac56b-Paper-Conference.pdf`.

Jacob Cohen. A coefficient of agreement for nominal scales. *Educational and psychological measurement*, 20(1):37–46, 1960.

Bradley Efron. Bootstrap methods: another look at the jackknife. In *Breakthroughs in statistics: Methodology and distribution*, pp. 569–593. Springer, 1992.

Orlena CZ Gotel and CW Finkelstein. An analysis of the requirements traceability problem. In *Proceedings of IEEE international conference on requirements engineering*, pp. 94–101. IEEE, 1994.

Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Yu, Xinying Song, and Denny Zhou. Large language models cannot self-correct reasoning yet. In *International Conference on Learning Representations*, volume 2024, pp. 32808–32824, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/file/8b4add8b0aa8749d80a34ca5d941c355-Paper-Conference.pdf.

Naman Jain, Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. In *International Conference on Learning Representations*, volume 2025, pp. 58791–58831, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/94074dd5a072d28ff75a76dabed43767-Paper-Conference.pdf.

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*, volume 2024, pp. 54107–54157, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/file/edac78c3e300629acfe6cbe9ca88fb84-Paper-Conference.pdf.

Brendan King and Jeffrey Flanigan. Dialogue swe-bench: A benchmark for dialogue-driven coding agents, 2026. URL https://arxiv.org/abs/2606.13995.

Klaus Krippendorff. Content analysis. *International encyclopedia of communication*, (226):403–407, 1989.

Aayush Kumar, Yasharth Bajpai, Sumit Gulwani, Gustavo Soares, and Emerson Murphy-Hill. Why ai agents still need you: Findings from developer-agent collaborations in the wild. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 432–444. IEEE Press, 2025. doi: 10.1109/ASE63991.2025.00043. URL https://doi.org/10.1109/ASE63991.2025.00043.

Dawei Li, Bohan Jiang, Liangjie Huang, Alimohammad Beigi, Chengshuai Zhao, Zhen Tan, Amrita Bhattacharjee, Yuxuan Jiang, Canyu Chen, Tianhao Wu, Kai Shu, Lu Cheng, and Huan Liu. From generation to judgment: Opportunities and challenges of LLM-as-a-judge. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 2757–2791. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.emnlp-main.138. URL https://aclanthology.org/2025.emnlp-main.138/.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-refine: Iterative refinement with self-feedback. In *Advances in Neural Information Processing Systems*, volume 36, pp. 46534–46594. Curran Associates, Inc., 2023. doi: 10.52202/075280-2019. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/91edff07232fb1b55a505a9e9f6c0ff3-Paper-Conference.pdf.

Adyasha Maharana, Dong-Ho Lee, Sergey Tulyakov, Mohit Bansal, Francesco Barbieri, and Yuwei Fang. Evaluating very long-term conversational memory of LLM agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 13851–13870. Association for Computational Linguistics, 2024. doi: 10.18653/v1/2024.acl-long.747. URL https://aclanthology.org/2024.acl-long.747/.

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. *Proc. ACM Softw. Eng.*, 1(FSE), 2024. doi: 10.1145/3660810. URL https://doi.org/10.1145/3660810.

Niels Mündler, Mark Niklas Müller, Jingxuan He, and Martin Vechev. Swt-bench: Testing and validating real-world bug-fixes with code agents. In *Advances in Neural Information Processing Systems*, volume 37, pp. 81857–81887. Curran Associates, Inc., 2024. doi: 10.52202/079017-2601. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/94f093b41fc2666376fb1f667fe282f3-Paper-Conference.pdf.

Jane Pan, Ryan Shar, Jacob Pfau, Ameet Talwalkar, He He, and Valerie Chen. When benchmarks talk: Re-evaluating code LLMs with interactive feedback. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 24672–24700. Association for Computational Linguistics, 2025a. doi: 10.18653/v1/2025.findings-acl.1267. URL https://aclanthology.org/2025.findings-acl.1267/.

Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. Training software engineering agents and verifiers with swe-gym. In *Proceedings of the 42nd International Conference on Machine Learning*, ICML'25. JMLR.org, 2025b.

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, 2026. URL https://arxiv.org/abs/2607.21217.

Hossein A. Rahmani, Xi Wang, Yue Feng, Qiang Zhang, Emine Yilmaz, and Aldo Lipani. A survey on asking clarification questions datasets in conversational systems. In Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki (eds.), *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 2698–2716, Toronto, Canada, July 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.acl-long.152. URL https://aclanthology.org/2023.acl-long.152/.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 36, pp. 8634–8652. Curran Associates, Inc., 2023. doi: 10.52202/075280-0377. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/1b44b878bb782e6954cd888628510e90-Paper-Conference.pdf.

Haoran Tan, Zeyu Zhang, Chen Ma, Xu Chen, Quanyu Dai, and Zhenhua Dong. MemBench: Towards more comprehensive evaluation on the memory of LLM-based agents. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 19336–19352. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.findings-acl.989. URL https://aclanthology.org/2025.findings-acl.989/.

Sanidhya Vijayvargiya, Xuhui Zhou, Akhila Yerukola, Maarten Sap, and Graham Neubig. Ambig-swe: Interactive agents to overcome underspecificity in software engineering. In *International Conference on Learning Representations*, volume 2026, pp. 146716–146737, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/file/ed4deb2ba9d4640fb6a000c688c2972e-Paper-Conference.pdf.

Ruiqi Wang, Jiyu Guo, Cuiyun Gao, Guodong Fan, Chun Yong Chong, and Xin Xia. Can llms replace human evaluators? an empirical study of llm-as-a-judge in software engineering. *Proc. ACM Softw. Eng.*, 2(ISSTA), 2025a. doi: 10.1145/3728963. URL https://doi.org/10.1145/3728963.

Xin Wang, Liangtai Sun, Yaoming Zhu, Shuang Zhou, Jiaxing Liu, Fengjiao Chen, Lin Qiu, Xuezhi Cao, Xunliang Cai, Licheng Zhang, and Zhendong Mao. Asuka-bench: Benchmarking code agents on underspecified user intent and multi-round refinement, 2026. URL https://arxiv.org/abs/2606.05920.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill

Qian, Daniel 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, 2025b. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/a4b6ad6b48850c0c331d1259fc66a69c-Paper-Conference.pdf.

Di Wu, Hongwei Wang, Wenhao Yu, Yuwei Zhang, Kai-Wei Chang, and Dong Yu. Longmemeval: Benchmarking chat assistants on long-term interactive memory. In *International Conference on Learning Representations*, volume 2025, pp. 86809–86836, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/d813d324dbf0598bbdc9c8e79740ed01-Paper-Conference.pdf.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Demystifying llm-based software engineering agents. *Proc. ACM Softw. Eng.*, 2(FSE), 2025. doi: 10.1145/3715754. URL https://doi.org/10.1145/3715754.

Derong Xu, Yi Wen, Pengyue Jia, Yingyi Zhang, Wenlin Zhang, Yichao Wang, Huifeng Guo, Ruiming Tang, Xiangyu Zhao, Enhong Chen, and Tong Xu. From single to multi-granularity: Toward long-term memory association and selection of conversational agents. In *International Conference on Learning Representations*, volume 2026, pp. 91581–91617, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/file/93e444b4a1a64815631c3c74977ee200-Paper-Conference.pdf.

Wujiang Xu, Zujie Liang, Kai Mei, Hang Gao, Juntao Tan, and Yongfeng Zhang. A-mem: Agentic memory for llm agents. In *Advances in Neural Information Processing Systems*, volume 38, Main Conference, pp. 17577–17604. Curran Associates, Inc., 2025. doi: 10.52202/085713-0593. URL https://proceedings.neurips.cc/paper_files/paper/2025/file/19909c36f51abc4856b4560aff3d36d6-Paper-Conference.pdf.

John Yang, Akshara Prabhakar, Karthik Narasimhan, and Shunyu Yao. Intercode: Standardizing and benchmarking interactive coding with execution feedback. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (eds.), *Advances in Neural Information Processing Systems*, volume 36, pp. 23826–23854. Curran Associates, Inc., 2023. doi: 10.52202/075280-1035. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/4b175d846fb008d540d233c188379ff9-Paper-Datasets_and_Benchmarks.pdf.

John Yang, Carlos 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. Curran Associates, Inc., 2024. doi: 10.52202/079017-1601. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/5a7c947568c1b1328ccc5230172e1e7c-Paper-Conference.pdf.

John Yang, Kilian Lieret, Carlos Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. Swe-smith: Scaling data for software engineering agents. In *Advances in Neural Information Processing Systems*, volume 38, Main Conference. Curran Associates, Inc., 2025. doi: 10.52202/085713-3239. URL https://proceedings.neurips.cc/paper_files/paper/2025/file/8b86cf5ace600c48fd188efbb8dedec8-Paper-Datasets_and_Benchmarks_Track.pdf.

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*, volume 2025, pp. 9965–10017, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/1b126cc38b8638e07bef37e7b2bb72bf-Paper-Conference.pdf.

Daoguang Zan, Zhirong Huang, Wei Liu, Hanwu Chen, Shulin Xin, Linhao Zhang, Qi Liu, Li Aoyan, Lu Chen, Xiaojian Zhong, Siyao Liu, Yongsheng Xiao, Liangqiang Chen, Yuyu Zhang, Jing Su, Tianyu Liu, RUI LONG, Ming Ding, and liang xiang. Multi-swe-bench: A multilingual benchmark for issue resolving. In *Advances in Neural Information Processing*

Systems, volume 38, Main Conference. Curran Associates, Inc., 2025. doi: 10.52202/085713-2111. URL https://proceedings.neurips.cc/paper_files/paper/2025/file/5afa9cb1e917b898ad418216dc726fbd-Paper-Datasets_and_Benchmarks_Track.pdf.

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, 2026a. URL https://arxiv.org/abs/2608.09072.

Xuhui Zhou, Valerie Chen, Zora Zhiruo Wang, Graham Neubig, Maarten Sap, and Xingyao Wang. TOM-SWE: User mental modeling for software engineering agents. In *Forty-third International Conference on Machine Learning*, 2026b. URL https://openreview.net/forum?id=HeLa76rJuD.

Yuxuan Zhu, Tengjun Jin, Yada Pruksachatkun, Andy Zhang, Shu Liu, Sasha Cui, Sayash Kapoor, Shayne Longpre, Kevin Meng, Rebecca Weiss, Fazl Barez, Rahul Gupta, Jwala Dhamala, Jacob Merizian, Mario Giulianelli, Harry Coppock, Cozmin Ududec, Antony Kellermann, Jasjeet Sekhon, Jacob Steinhardt, Sarah Schwettmann, Arvind Narayanan, Matei A Zaharia, Ion Stoica, Percy Liang, and Daniel Kang. Establishing best practices in building rigorous agentic benchmarks. In *Advances in Neural Information Processing Systems*, volume 38, Main Conference. Curran Associates, Inc., 2025. doi: 10.52202/085713-5547. URL https://proceedings.neurips.cc/paper_files/paper/2025/file/f316275b44ee2de533102913828a8107-Paper-Datasets_and_Benchmarks_Track.pdf.

Mingchen Zhuge, Changsheng Zhao, Dylan R. Ashley, Wenyi Wang, Dmitrii Khizbullin, Yunyang Xiong, Zechun Liu, Ernie Chang, Raghuraman Krishnamoorthi, Yuandong Tian, Yangyang Shi, Vikas Chandra, and Jürgen Schmidhuber. Agent-as-a-judge: evaluate agents with agents. In *Proceedings of the 42nd International Conference on Machine Learning*, ICML'25. JMLR.org, 2025.

A Appendix guide and evidence sources

The appendix connects the benchmark's construction, evaluation protocol, and results to the review package. Appendices B–F specify target construction, runtime behavior, recovery judging, metric conventions, and experimental setup. Appendices G–M expand the main comparisons through uncertainty estimates, request-view effects, interaction analyses, revision dynamics, and task-level results. Appendix N supplies the prompt library, and Appendix O gives the reproduction entry points.

Evidence sources. Statistical tables and figures are regenerated from 11,700 frozen episode records, requirement-level judge votes, recorded execution outcomes, baseline results, task metadata, and anonymous human labels. The construction and runtime descriptions are tied to the released workflow source. No new model generations enter these analyses. The appendix renderer checks shared estimates against the main-text inputs before formatting. Matching cells therefore use the same observations and aggregation rules. Appendix objects use A-prefixed numbering. The distinction between recomputing recorded measurements and executing a new evaluation is specified in Appendix O.

B Benchmark construction details

This appendix expands Section 3.1 by describing how source interactions become personas and task kernels, how the two drafting passes produce frozen targets, which automated gates every task must clear, and how the three request views are derived. The prompts driving each stage appear in Appendix N.5.

B.1 Provenance of reference targets and human labels

The term *gold* in the construction code and archived prompts denotes a fixed evaluation reference. It does not imply that the original user supplied or approved every scored behavior. The reference objects have the following origins.

- **Persona implications.** The persona model distills preferences from eligible earlier sessions and associates them with evidence identifiers. These construction annotations seed requirements assigned to profile/history recovery.
- **Specifications, requirement catalogs, and reference answers.** The task constructor drafts these from the selected kernel and admissible context, then expands and validates the target. The fields `clarifying_question` and `ground_truth_answer` are generated reference pairs, not newly collected user responses. Engineering defaults and operational details are choices of the constructed target.
- **Reference code and expected test behavior.** The construction pipeline generates implementations and executable checks, reconciles requirement bindings, and verifies reference execution. These establish internal consistency with the target, rather than independent user confirmation.
- **Human validation labels.** Two annotators assess 90 construction packets and independently label entailment for 360 requirement–brief pairs. These labels evaluate construction quality and judge agreement. They are separate from the model-generated requirement catalog and are not feedback from the source users.

The fixed User Simulator subsequently answers against these references. Its replies are evaluation-time model outputs under the policy in Appendix C.4. This distinction limits claims of intent recovery to the evidence-grounded target defined by the benchmark.

B.2 Source interactions, personas, and admissible evidence

Each of the 101 pseudonymized users contributes retained software-development interactions. Candidate sessions are first normalized and filtered, then scored and summarized per session. A selection model chooses sessions whose incomplete programming intent can be converted into a self-contained Python task (prompts `SYS_SELECT_SINGLE` and `SYS_SELECT_REPO` in Appendix N.5). For each selected kernel a construction cutoff is fixed at the source session's timestamp, and all admissible evidence must come from strictly earlier sessions by the same user.

The persona builder uses SYS_PERSONA to produce two artifacts for different roles. The *Producer-facing view*, a rough public profile plus a de-identified interaction history, is what evaluated models read at run time. It presents contextual evidence from which preferences must be inferred, rather than an explicit list of the scored requirements. The *gold implications* are model-distilled preferences inferred from the admissible earlier sessions, with concrete code-rule hints and per-session evidence. They are construction annotations, not user-authored labels or new user feedback. They are visible only during construction, where they seed hidden requirements recoverable from context. Every history-derived implication requires endorsement from at least two distinct pre-cutoff sessions, and a recoverability audit using SYS_RECOVER re-derives each history-routed requirement from the Producer-facing view alone, rerouting unsupported profile/history requirements to clarification or excluding the task when the required channel support cannot be retained.

B.3 Two-pass target drafting

Pass one uses SYS_DESIGN to draft the complete frozen target from the kernel and admissible context, including the full specification, callable contract, reference implementation, executable tests, and the core atomic requirements, with recovery labels and criticality assigned to hidden requirements. Clarification targets additionally carry a reference question and a construction-generated reference answer. The stored field ground_truth_answer contains the target behavior drafted in this pass. It is not a transcript of an answer collected from the original user. Pass two uses SYS_EXPAND to mine the reference implementation for additional observable behavioral decisions the sparse request leaves open, including ordering, tie-breaking, deduplication, empty-input handling, and error semantics, and emits each as a further atomic requirement. Requirements from both passes are bound to executable tests. A companion naive solution generated by SYS_NAIVE implements the sparse request, and the construction pipeline checks which hidden requirements its outputs fail. Test augmentation through SYS_AUGMENT adds inputs for designated requirements that lack discrimination. Track-specific acceptance criteria combine requirement coverage, reference execution, and naive-solution discrimination. This diagnostic is relative to the generated naive solution, rather than every possible implementation. Repository-track drafting uses parallel prompts that target a multi-module package with a declared architecture instead of a single callable, including behavior-preserving growth of the reference package to repository scale.

B.4 Automated acceptance gates

Table A1 summarizes the acceptance checks across construction and view derivation. Contract, binding, and reference checks establish the frozen target. Leakage and hint checks also inspect the derived views. Repairs, exclusions, and binding reconciliations are recorded during construction. The gates check internal consistency. Semantic faithfulness and recoverability are assessed by the human audit of Section 4.1, whose per-check results appear in Table 3A.

B.5 Inventory statistics

The frozen dataset counts referenced in Section 3.2 are collected in Table A2. The 100 additional repository package checks verify installability and import structure. They are retained as execution diagnostics and are excluded from all requirement-level metrics. Task families (Figure 2) comprise 168 feature tasks, 52 bug fixes, 29 operations tasks, 18 research workflows, 17 refactorings, and 16 other tasks.

Every task carries at least one clarification target and at least one profile/history target, ensuring support for both channels on every task. Channel proportions describe the constructed mix, not real-world source prevalence.

B.6 Request-view derivation

The three views are derived from the frozen target by removal and controlled re-addition of information. **L0** applies prompt SYS_FUZZY: starting from the full specification, information is *only removed*, so the sparse request must be a strict subset of the specification, while the callable contract remains stated, along with the repository architecture and output shape when applicable, since those are given contracts rather than hidden intent. **L1** and **L2** are produced by the graded-hint prompt

Table A1: **Automated acceptance gates.** Checks span target construction and request-view derivation. Failures trigger repair, rerouting, or exclusion under the relevant stage policy.

Gate	What it verifies
Contract–reference consistency	The reference implementation satisfies the released callable contract (signature, entry point, module architecture on the repository track).
Requirement–test consistency	Every atomic requirement has a nonempty set of bound tests, and every binding resolves to an executable case.
Reference execution	The reference passes its bound tests, including expected exceptions when error behavior is specified.
Determinism	Repeated executions produce identical outputs. All external data and time enter through arguments.
Purity	Standard library only. No network, filesystem side effects, or nondeterministic sources.
Naive-solution discrimination	Track-specific checks require hidden-rule failures by the sparse-request naive solution. Designated undiscriminated requirements trigger test augmentation.
Temporal separation	All evidence sessions precede the construction cutoff. The source session and related sessions are excluded from admissible context.
De-identification	Source-specific identifying context is removed. Export checks screen released artifacts for identifying traces.
Leakage	The L0 request does not state hidden requirements (prompt `SYS_LEAK`). L2 hints avoid literal, test-checkable tokens (prompt `SYS_HINT_LEAK`).
Track structure	Single-file tasks expose one callable. Repository tasks expose the declared multi-module package layout and public symbols.

Table A2: **Frozen inventory.** All counts fixed before any request view is derived. Left: per-track structure. Right: the pooled requirement catalog.

Per track			**Pooled catalog and users**	
	Single-file	Repository		
Tasks	200	100	Atomic requirements	8,273
Source users	92	44	surface	896
Hidden reqs. per task	15–28	24–75	hidden	7,377
Executable tests	6,450	6,468	clarification	3,948 (53.5%)
requirement-bound	12,818 pooled		profile/history	640 (8.7%)
package checks	0	100	default	2,789 (37.8%)
Tests per task	20–49	40–134	Source users total (35 in both tracks)	101

`SYS_HINT`: L1 *names* an unresolved decision dimension without choosing a value. L2 additionally commits to a qualitative direction, such as ascending versus descending, raise versus return, or keep-first versus keep-last, while targeting the omission of literal answers such as exact field names, thresholds, counts, error-type names, and sort keys. A dedicated audit using `SYS_HINT_LEAK` flags hints that expose such literal tokens, and the human audit of Table 3A independently re-examines L1 non-directionality and L2 non-literalness, recording exceptions. Only clarification and default targets are eligible for hints. History-derived targets remain reachable through admissible context alone, and cross-view reconciliation verifies that all three views bind to the identical requirement set, so L2 remains an incomplete request rather than an oracle specification.

B.7 Worked construction case

Task `s3ts_020`, released as *Product Pipeline Normalization and Storage Planning*, illustrates the construction workflow using actual benchmark artifacts. The account below paraphrases the released request, history, requirement catalog, hints, and tests. It is not a reconstruction of the withheld raw conversation or a new evaluation episode.

1. Select a kernel and admissible context. The released opening request asks for a function that cleans scraped products grouped by category and returns products, a progress log, and a temporary-file placeholder. The task-local history records repeated expectations that browser automation run without a visible interface. The constructor uses this evidence to support a preference for headless operation. The exact validation behavior in the target remains a construction choice, rather than a verbatim historical instruction.

2. Draft and expand the target. The callable contract is `process_products(products_by_category, config)`. The frozen catalog con-

tains one surface requirement and 21 hidden requirements, comprising two profile/history targets, eight clarification targets, and eleven defaults. Table A3 shows how selected decisions receive different recovery labels and executable bindings. The two-pass procedure fixes the main behavior and then exposes further decisions, such as literal rather than regular-expression interpretation of blacklist phrases.

Table A3: **Selected requirements from task `s3ts_020`.** Statements are shortened for readability. Test identifiers and recovery labels follow the released catalog.

ID	Recovery source	Frozen target behavior	Bound tests
`r2`	Profile/history	Reject a supplied browser mode other than `headless` with `ValueError`.	`test_r2_0`, `test_r2_1`
`rx23_9`	Clarification	Interpret blacklist phrases as literal strings, including characters with special meaning in regular expressions.	`test_rx23_9_0`, `test_rx23_9_1`
`r12`	Default	Return an empty dictionary when product specifications are missing or `None`.	`test_r12_0`, `test_r12_1`

3. Bind and validate behavior. The released task has 40 executable tests. The bindings above distinguish behavioral recovery from implementation. For example, a brief may specify literal blacklist matching, but the artifact realizes `rx23_9` only if both bound tests pass. The catalog's answer for this requirement is a construction-generated reference describing literal, case-insensitive matching. It is available to the simulator, not a record of a live clarification from the source user. Reference execution and binding checks verify the frozen target before Producer evaluation.

4. Derive matched views. L0 supplies the function signature and output schema without resolving the hidden decisions. For `rx23_9`, L1 names the choice between exact text and special patterns. L2 guides the agent toward a fixed sequence of characters. For `r12`, L1 identifies missing product specifications as an unresolved case, and L2 suggests an empty container. These descriptions paraphrase `fuzzy_variants/hints.json`. History-routed `r2` has no hint and remains tied to prior context. The requirement catalog, reference, and tests are shared across all three views.

All corresponding files are under `dataset/single/tasks/s3ts_020/`. The case demonstrates how source-supported preferences and constructed behavioral choices become one scored target, while preserving their distinct origins.

C Producer runtime and interaction protocol

An episode consists of evidence gathering, an initial Code Agent call, and Producer-directed review. The same action interface and budgets apply to all Producers. Role prompts appear in Appendices N.1–N.3.

C.1 Action interface and budgets

The Producer is prompted to emit one JSON action with a short `thought` field. The runtime executes the action and returns its observation. Gathering ends at `finalize` or its stopping condition. The brief text, rather than the separately logged requirements list, is sent to the coder. Review can continue after the fourth revision, using remaining questions or example runs, but cannot request another artifact version. It ends on `accept`, the total action limit, or the malformed-response stop rule. The current artifact is delivered in each case. Questions share one quota across gathering and review.

C.2 Observations, parsing, and service calls

The observation policy uses fixed truncation limits: accumulated profile/history state is rendered to at most 9,000 characters, code to 9,000 (single-file) or 13,000 (repository), and example-run results to 3,000. History search returns up to four sections with positive lexical overlap, weighting heading matches twice as heavily as body matches. The Producer sees neither the construction-generated requirement catalog nor hidden-test feedback.

Table A4: **Producer actions and configured budgets.** Lookup, question, example-run, and revision limits are shared within each episode.

Phase	Action	Semantics	Budget
Gather	`read_profile`	Read role, seniority, domains, and technology context.	
Gather	`search_history`	Retrieve history sections by lexical overlap. An empty or unmatched query returns the section index.	10 lookups
Gather	`read_history`	Read the available history sections.	
Both	`ask_user`	Submit one question to the User Simulator.	14 questions
Gather	`finalize`	Submit the implementation brief and a separately logged requirements list.	once
Review	`run_code`	Run self-authored example calls. Observe sandbox outputs, not hidden-test verdicts.	6 executions
Review	`request_revision`	Send feedback to the Code Agent and obtain the next artifact version.	4 rounds
Review	`accept`	End review and deliver the current artifact.	once
Gathering-step limit / total Producer-step limit			18 / 40

Producer responses allow up to 6,000 output tokens. Simulator matching and specification-answering calls allow 800 and 500. In the released runtime, Producer and Code Agent calls use a 300-second timeout, while simulator calls use the shared client's 120-second default. The shared client defaults to six attempts with bounded exponential backoff and permits an explicit retry override. An unparseable Producer response receives one same-model JSON-format retry. Persistent parse failures count toward the five-failure episode threshold. These handling rules are common across Producers.

C.3 Brief finalization and fallback

The nonempty `brief` field supplied at finalization becomes B_i. If gathering leaves no usable brief, the runtime concatenates the request view and accumulated question and answer dialogue to form a fallback. The recorded `finalized` flag is false in 1,420 of 11,700 episodes. These episodes remain in every applicable aggregate. This accounting evaluates the whole workflow rather than only episodes with explicit finalization. An empty `finalize` payload can also invoke the fallback, so the flag records explicit finalization rather than certifying nonempty brief content.

C.4 User Simulator policy

The fixed `DeepSeek-V4-Flash` simulator first matches a question to at most one hidden requirement from its identifier, source label, statement, and available reference answer generated during target construction as described in Appendix B.3. A question spanning several decisions can therefore receive credit for only one target. Clarification matches receive the matching call's task-grounded answer, with the stored construction answer as fallback. Neither is fresh feedback from the original user. Profile/history matches receive a fixed response directing the Producer to prior context. Default and unmatched questions invoke specification-grounded answering on up to 8,000 characters of the task specification. Its prompt asks for a concise answer to the question alone and refusal of requests to enumerate the whole specification. The runtime logs the matched identifier and channel. Metric computation validates clarification credit against the frozen catalog.

C.5 Code Agent interface

The fixed `Kimi-K2.7-Code` receives the brief and callable contract, without the original request or user history. It returns a module for single-file tasks or a package respecting the declared repository architecture, with output limits of 16,000 and 24,000 tokens, respectively. Each revision receives the original brief, previous artifact, and Producer feedback and returns a full replacement. If extraction yields no usable replacement, the previous artifact is retained, $A_j = A_{j-1}$. Hidden grading is recorded for analysis and is never returned as a Producer observation. The review package provides recorded version scores. The fresh-run workflow retains newly generated artifacts and their test outcomes.

D RECOVERY JUDGING AND VALIDATION

D.1 JUDGE PANEL AND VOTE HANDLING

GPT-5.5, Gemini-3.5-Flash, and DeepSeek-V4-Flash independently judge whether the pre-implementation brief entails each atomic requirement under the rubric in Appendix N.4. A positive decision requires a verbatim, contiguous supporting quote. The judging interface checks that the quote occurs in the brief. Recovery uses a majority of three votes. Judges receive no code, test outcomes, or other judges' decisions. The automated payload includes the Producer identifier. The separate human-validation protocol is identity-blinded. Missing or invalid panels remain unscored until resolved. The released record contains complete panels for all 11,700 episodes.

D.2 PANEL SENSITIVITY

Pairwise agreement on the 287,703 hidden-requirement records ranges from 79.9% to 91.7%. Leaving out each judge and requiring agreement of the remaining two preserves the pooled thirteen-Producer IR ranking with Spearman $\rho \geq 0.9615$. Using either remaining judge's positive vote also gives $\rho \geq 0.9780$. These complementary thresholds show that the ranking is stable across the examined panel rules.

All three judges also occur in the evaluated roster. We report a descriptive family contrast with an explicit common denominator. For judge j and Producer m, let s_{jm} be the mean hidden-requirement positive-vote rate over tasks and views (tracks pooled 2:1), and let $d_{jm} = s_{jm} - \frac{1}{2}\sum_{j' \neq j} s_{j'm}$. The contrast is mean d_{jm} on the judge's own-family Producers minus its mean on the remaining Producers, weighting Producers equally. On the full roster, the contrasts are -0.3 pp for GPT, $+1.0$ pp for Gemini, and -0.1 pp for DeepSeek. Family groups contain the two GPT, two Gemini, and three DeepSeek Producers in Table 2. These vote-rate differences also reflect differences in brief content. Human agreement below provides the external validation of recovery judgments.

D.3 HUMAN VALIDATION PROTOCOLS

Construction audit. Two annotators each reviewed 90 packets (60 single-file and 30 repository), including 53 flagged by construction-time diagnostics. This risk-enriched sample supports targeted examination of construction quality. Each packet contains source evidence, the target, requirements, and all three views. The eight checks are those in Table 3A. Binary checks enter as task pass rates. Fraction-valued checks are averaged within packets and then across packets. Mean compliance is at least 94.4% for both annotators on every check, and all 90 packets pass temporal separation and reference validity for both. Fraction-valued annotations retain per-packet counts, sometimes with different denominators, so they support each annotator's compliance estimate rather than item-level inter-annotator agreement.

Recovery-judge validation. The recorded sampling design selects 360 hidden-requirement–brief pairs: three from each of 120 strata defined by two tracks, ten Producers, three views, and the original unanimous/non-unanimous label. The sample contains 180/32/50/98 items with 3/2/1/0 positive panel votes. Majority voting therefore changes 32 labels relative to unanimity. Two annotators independently label entailment while blinded to Producer identity, panel votes, and each other's labels. Reported panel comparisons use each annotator's retained labels separately.

The majority panel agrees with annotators A and B on 91.9% and 90.8% of items, with F1 scores of 93.3% and 92.1% (Table 3B). Agreement against A is 92.0% under the original sampling weights. Agreement between annotators is 95.0%, with nominal Krippendorff's $\alpha = 0.896$ and Cohen's $\kappa = 0.896$. Against A, unanimity gives 98.9% precision and 79.8% recall, compared with 95.8% and 91.0% for majority voting. The majority rule provides higher coverage while retaining high precision. A separately recorded GPT-5.5 pilot was used only to check the annotation workflow and contributes no human labels or validation scores.

E METRIC CONVENTIONS AND AGGREGATION

This appendix specifies the edge cases and denominators needed to reproduce the five metrics of Section 3.3.

E.1 EPISODE-LEVEL CONVENTIONS

Execution and DC. A requirement is realized only when all its bound tests pass. Failed, skipped, unrun, timed-out, or unimportable cases contribute zero to the affected requirements. Requirements have equal weight. The 100 additional repository package checks are reported separately. Stored episode DC and version scores have four decimal places, and signed ReviewGain uses those retained values.

IR and RIR. Majority recovery is defined on hidden requirements only. The recovered set is fixed by the actual pre-implementation brief, including a runtime fallback when applicable. Later feedback does not enlarge it. RIR is undefined for an empty recovered set and is omitted only from that metric's aggregation. Twelve Producers have at least 590/600 eligible single-file and 290/300 eligible repository episodes. Gemini-3.5-Flash has 399/600 and 199/300. Reading IR alongside RIR distinguishes broad recovery from successful implementation of a smaller recovered subset.

INT-F1. With distinct validated clarification hits h_i, submitted questions q_i, and fixed clarification support $c_i > 0$, $\text{INT-F1}_i = 2h_i/(q_i + c_i)$. Questions in either phase count. Repeats, off-route matches, and unmatched questions increase q_i without additional clarification credit. Asking none gives zero. Credit requires a matched identifier and clarification label consistent with the frozen catalog.

ReviewGain. $\text{RG}_i = \text{DC}_i(A_F) - \text{DC}_i(A_0)$ retains positive, zero, and negative changes. No revision gives zero regardless of whether the Producer explicitly accepts the first artifact.

E.2 AGGREGATION AND UNCERTAINTY

For each Producer and track, the main five metrics average eligible views within each task and then average contributing tasks equally. Complete pooled summaries weight single-file and repository tracks 2:1. Per-view summaries average tasks at that view. Level contrasts use within-task differences on the intersection of eligible tasks. Pooled contrasts average those eligible pairs, so RIR's effective track weights can differ from 2:1.

The main metric intervals use 10,000 percentile-bootstrap draws of tasks within each track, with seed 20260921. The appendix uses the same draw indices across Producers and metrics for paired comparisons. Source-user sensitivity resamples users within track and carries along all their tasks: 92 users in single-file and 44 in repository, with 35 shared users and 101 distinct users overall. Intervals describe variation over the sampled tasks or users conditional on the retained trajectories. They do not estimate repeated-generation variability. No missing observation is replaced by zero.

Supporting summaries use their stated units: channel means average supported tasks at L0. Question trajectories first average the thirteen Producers within each task. Burden ratios pool question or token counts. Revision cohorts pool episodes of the same final depth. Their exact denominators are specified in Appendices J and K.

F EXPERIMENTAL SETUP AND ACCOUNTING

Models and shared services. Table 2 names all thirteen Producers. The released configuration maps their record identifiers to service names and fixes the Code Agent, User Simulator, judge panel, and budgets. Scores describe the services accessed for the recorded evaluation. GPT-5.5, Gemini-3.5-Flash, and DeepSeek-V4-Flash serve as both judges and Producers. DeepSeek-V4-Flash is also the simulator, and DeepSeek-V4-Pro is the construction drafter. Panel sensitivity is reported in Appendix D.2.

Table A5 lists every model that participates in the protocol, its role, and the released service identifier. Only the Code Agent is reserved for non-Producer use; the remaining non-Producer roles

reuse services that also appear in the Producer roster. Because the submission is anonymous, models are named by their public service identifiers rather than by provider endpoints, and the released configuration records these identifiers verbatim (`configs/main_experiment.json`, `reproduction/data/provenance.json`).

Table A5: **Model roster and shared services.** Service identifiers follow the released configuration. Roles: P = Producer, C = fixed Code Agent (also the backend for both coder references), S = User Simulator, J = recovery judge, D = construction drafter. `Kimi-K3` (a Producer) and `Kimi-K2.7-Code` (the Code Agent) are distinct services in the same family.

Model	Role(s)	Released service identifier
Claude-Fable-5	P	`claude-fable-5`
Claude-Opus-4.8	P	`aws.claude-opus-4.8`
Gemini-3.8-Flash	P	`gemini-3.8-flash`
Gemini-3.5-Flash	P; J	`gemini-3.5-flash`
GPT-5.6-Terra	P	`gpt-5.6-terra`
GPT-5.5	P; J	`gpt-5.5`
DeepSeek-V4.1-Flash	P	`deepseek-v4.1-flash`
DeepSeek-V4-Flash	P; S; J	`deepseek-v4-flash`
DeepSeek-V4-Pro	P; D	`deepseek-v4-pro`
Qwen3.5-Plus	P	`qwen3.5-plus`
Kimi-K3	P	`kimi-k3`
GLM-5.2	P	`glm-5.2`
MiniMax-M3	P	`MiniMax-M3`
Kimi-K2.7-Code	C	`kimi-k2.7-code`

Episode accounting. There is one retained trajectory for every combination of Producer, task, and view, totaling $13 \times 300 \times 3 = 11,700$ episodes. The released votes and recorded requirement outcomes reconcile with all five metrics. The provenance file additionally records upstream brief-integrity checks against the original transcripts. Those transcripts are not part of the public evidence. The 1,420 episodes without explicit finalization remain in the evaluation under the fallback policy in Appendix C.3. Call-level retries and format handling are shared runtime rules, rather than selection among several completed evaluation trajectories.

Comparison scope. Fixed role services and request-view targets support controlled comparisons among Producers. The two tracks share a protocol but contain different tasks, requirement inventories, and artifact formats. Track differences therefore describe those settings jointly. Changing the coder or simulator defines a new evaluation configuration. The release exposes these choices explicitly for future comparisons.

G Main results with confidence intervals

Tables A6 and A7 pair every main-table estimate with its 95% task-bootstrap interval. Bold and underline retain the point-estimate ordering. The intervals make the precision of each comparison visible, including the wider repository DC and RIR intervals.

Table A8 directly compares the best and second-best Producer in each metric. On single-file tasks, the paired intervals exclude zero for IR, RIR, INT-F1, and ReviewGain. The DC contrast is $+0.3$ pp with interval $[-1.7, +2.2]$. The repository contrasts also include zero. Resampling source users instead of tasks preserves all ten separation judgments, with displayed endpoints changing by at most 0.8 pp. These are descriptive, unadjusted intervals for leaders selected from the same evaluation. They are not simultaneous guarantees over all model comparisons.

H Results by request view

Tables A9 and A10 report all five metrics separately at L0, L1, and L2, alongside the request-only coder's DC on the same views. These values complement the across-view averages in Table 2 and the paired contrasts in Figure 4.

Table A6: **Single-file track: five metrics with 95% task-bootstrap intervals** (200 tasks, three views pooled; 10,000 paired track-stratified draws). Point estimates repeat Table 2; IR, DC, RIR, INT-F1 in %, RG in pp. Best per column **bold**, second underlined.

Producer	IR	DC	RIR	INT-F1	RG
Claude-Fable-5	**72.1** [70.4, 73.6]	62.3 [59.6, 64.9]	72.4 [69.5, 75.3]	**39.5** [38.0, 40.8]	4.9 [3.4, 6.6]
Claude-Opus-4.8	61.0 [59.1, 62.9]	52.5 [49.6, 55.3]	63.4 [60.0, 66.6]	34.5 [33.2, 35.8]	5.6 [4.2, 7.1]
Gemini-3.8-Flash	65.2 [63.3, 67.0]	**63.6** [61.2, 66.0]	**76.8** [74.2, 79.4]	29.2 [27.7, 30.8]	5.9 [4.4, 7.5]
Gemini-3.5-Flash	31.4 [28.9, 33.9]	48.5 [45.8, 51.2]	58.8 [54.4, 63.2]	20.8 [19.3, 22.4]	**19.5** [17.3, 21.8]
GPT-5.6-Terra	68.0 [66.1, 69.9]	57.7 [54.6, 60.8]	67.7 [64.3, 71.0]	32.3 [31.0, 33.7]	11.4 [9.3, 13.6]
GPT-5.5	69.2 [67.5, 70.9]	59.2 [56.2, 62.1]	69.7 [66.4, 72.8]	33.4 [32.0, 34.8]	6.2 [4.6, 8.0]
DeepSeek-V4.1-Flash	69.4 [67.6, 71.1]	63.3 [60.6, 65.9]	73.7 [70.8, 76.5]	34.6 [33.4, 35.9]	8.4 [6.7, 10.2]
DeepSeek-V4-Flash	50.1 [48.2, 51.9]	48.2 [45.6, 50.8]	61.4 [58.2, 64.6]	32.0 [30.6, 33.4]	9.4 [7.8, 11.2]
DeepSeek-V4-Pro	60.4 [58.4, 62.5]	52.8 [49.9, 55.7]	63.6 [60.3, 67.0]	28.8 [27.5, 30.1]	5.4 [4.0, 6.9]
Qwen3.5-Plus	57.9 [56.1, 59.8]	55.0 [52.3, 57.7]	67.2 [64.1, 70.2]	33.1 [31.7, 34.5]	6.0 [4.5, 7.7]
Kimi-K3	61.2 [59.4, 63.1]	51.9 [49.1, 54.6]	62.8 [59.7, 65.9]	31.8 [30.5, 33.2]	6.0 [4.6, 7.6]
GLM-5.2	54.8 [52.8, 56.9]	52.3 [49.6, 55.0]	66.3 [63.1, 69.4]	29.5 [28.2, 30.8]	6.8 [5.2, 8.6]
MiniMax-M3	58.6 [56.8, 60.4]	44.8 [42.2, 47.4]	57.5 [54.4, 60.7]	26.5 [24.9, 28.0]	4.9 [3.5, 6.3]

Table A7: **Repository track: five metrics with 95% task-bootstrap intervals** (100 tasks, three views pooled; 10,000 paired track-stratified draws). Point estimates repeat Table 2; IR, DC, RIR, INT-F1 in %, RG in pp. Best per column **bold**, second underlined.

Producer	IR	DC	RIR	INT-F1	RG
Claude-Fable-5	**64.5** [61.8, 66.9]	**50.7** [46.0, 55.3]	51.3 [45.2, 57.4]	**28.7** [26.8, 30.5]	8.4 [5.4, 11.6]
Claude-Opus-4.8	56.6 [54.2, 58.9]	46.9 [42.4, 51.3]	48.5 [42.7, 54.3]	25.1 [23.2, 27.0]	8.3 [5.6, 11.1]
Gemini-3.8-Flash	54.3 [51.5, 56.9]	50.4 [45.8, 55.0]	**54.0** [47.8, 60.0]	21.7 [19.7, 23.7]	8.9 [6.3, 11.6]
Gemini-3.5-Flash	28.3 [25.3, 31.3]	44.4 [40.0, 48.8]	44.5 [37.7, 51.4]	13.9 [12.1, 15.7]	**15.2** [12.5, 17.8]
GPT-5.6-Terra	62.3 [59.9, 64.6]	46.1 [41.2, 50.9]	47.2 [41.0, 53.4]	24.5 [22.7, 26.2]	11.8 [9.2, 14.5]
GPT-5.5	63.1 [60.8, 65.3]	47.5 [42.7, 52.2]	48.5 [42.3, 54.7]	25.0 [23.1, 26.9]	11.4 [8.3, 14.6]
DeepSeek-V4.1-Flash	62.7 [60.3, 65.0]	48.7 [43.9, 53.4]	50.2 [44.1, 56.2]	27.3 [25.5, 29.0]	9.0 [6.1, 12.0]
DeepSeek-V4-Flash	47.1 [44.6, 49.6]	43.3 [39.1, 47.5]	45.1 [39.4, 51.0]	23.9 [22.1, 25.7]	9.5 [7.3, 11.7]
DeepSeek-V4-Pro	58.2 [55.6, 60.8]	46.3 [41.5, 51.1]	46.2 [40.0, 52.5]	23.9 [22.3, 25.4]	9.0 [6.7, 11.5]
Qwen3.5-Plus	47.6 [45.2, 49.9]	46.0 [41.3, 50.6]	49.3 [43.3, 55.3]	23.3 [21.5, 25.1]	8.9 [6.4, 11.5]
Kimi-K3	57.1 [54.5, 59.5]	45.0 [40.6, 49.3]	45.4 [39.6, 51.2]	23.5 [21.7, 25.3]	10.3 [7.4, 13.3]
GLM-5.2	51.9 [49.2, 54.5]	46.4 [41.9, 51.1]	48.3 [42.1, 54.5]	21.5 [19.7, 23.4]	7.5 [5.0, 10.2]
MiniMax-M3	53.5 [51.1, 55.8]	38.5 [34.2, 42.8]	37.9 [32.4, 43.4]	20.5 [18.7, 22.4]	9.9 [7.4, 12.5]

Table A8: **Best-versus-second paired contrasts.** Within-task paired differences (pp) between the point-estimate leaders per column of Table 2, with 95% intervals from the same 10,000 paired task-bootstrap draws; "sep." marks intervals excluding zero. The last column repeats the interval under the source-user-cluster bootstrap (users resampled instead of tasks); every separation verdict is unchanged. Intervals are descriptive and unadjusted for leader selection.

Track	Metric	Best	Second	Δ	95% CI (task)	sep.	95% CI (user cluster)
Single-file	IR	Claude-Fable-5	DeepSeek-V4.1-Flash	+2.7	[+1.0, +4.3]	yes	[+0.8, +4.6]
Single-file	DC	Gemini-3.8-Flash	DeepSeek-V4.1-Flash	+0.3	[-1.7, +2.2]	no	[-1.6, +2.1]
Single-file	RIR	Gemini-3.8-Flash	DeepSeek-V4.1-Flash	+3.1	[+1.0, +5.3]	yes	[+1.0, +5.2]
Single-file	INT-F1	Claude-Fable-5	DeepSeek-V4.1-Flash	+4.8	[+3.2, +6.3]	yes	[+3.4, +6.3]
Single-file	RG	Gemini-3.5-Flash	GPT-5.6-Terra	+8.1	[+5.1, +11.2]	yes	[+5.3, +11.0]
Repository	IR	Claude-Fable-5	GPT-5.5	+1.4	[-0.8, +3.4]	no	[-1.3, +3.9]
Repository	DC	Claude-Fable-5	Gemini-3.8-Flash	+0.3	[-2.1, +2.6]	no	[-2.9, +2.8]
Repository	RIR	Gemini-3.8-Flash	Claude-Fable-5	+2.7	[-0.6, +6.0]	no	[-0.5, +6.2]
Repository	INT-F1	Claude-Fable-5	DeepSeek-V4.1-Flash	+1.4	[-0.2, +3.0]	no	[-0.6, +3.4]
Repository	RG	Gemini-3.5-Flash	GPT-5.6-Terra	+3.4	[-0.1, +6.9]	no	[-0.4, +7.3]

Every Producer exceeds the request-only coder's mean DC within each track and view. On single-file tasks, the reference improves from 20.5% at L0 to 29.4% at L2. Claude-Fable-5 improves from 54.8% to 71.5%, alongside IR rising from 55.5% to 86.5%. Across Producers, single-file L2–L0 IR increases span 26.5–42.8 pp, compared with 5.5–20.9 pp for DC. Thus richer requests consistently support recovery, while recovery and delivered behavior remain distinct outcomes.

INT-F1 peaks at L1 for all thirteen Producers on both tracks. Naming unresolved dimensions is associated with the best balance of distinct clarification hits and submitted questions under the fixed

Table A9: **Single-file track by request view.** All five metrics at each information level (L0 sparse request, L1 named dimensions, L2 qualitative direction); 200 tasks per view, scored on the same frozen targets; RIR uses only eligible episodes. IR, DC, RIR, INT-F1 in %, RG in pp; the final row gives the request-only coder DC on the identical views. Best per column **bold**.

	L0 (sparse request)					L1 (named dimensions)					L2 (qualitative direction)				
Producer	IR	DC	RIR	F1	RG	IR	DC	RIR	F1	RG	IR	DC	RIR	F1	RG
Claude-Fable-5	**55.5**	54.8	71.6	**34.2**	3.2	**74.1**	60.6	70.1	**48.3**	5.6	**86.5**	71.5	76.1	**35.9**	6.1
Claude-Opus-4.8	39.0	46.6	64.0	26.4	4.6	64.3	51.1	60.8	41.8	7.5	79.7	59.8	65.3	35.3	4.8
Gemini-3.8-Flash	50.3	**57.7**	**77.1**	27.0	4.2	64.1	**63.2**	**77.6**	32.7	6.4	81.1	69.9	75.8	28.0	7.0
Gemini-3.5-Flash	19.8	41.5	59.2	16.5	**13.3**	28.0	48.5	62.5	23.0	**19.8**	46.4	55.5	56.9	23.0	**25.4**
GPT-5.6-Terra	55.1	55.1	69.3	28.2	9.6	67.0	55.8	67.1	38.6	12.7	82.0	62.2	67.1	30.1	11.9
GPT-5.5	53.6	51.2	66.8	29.9	4.9	71.2	58.8	69.1	38.9	8.2	82.9	67.6	73.4	31.5	5.7
DeepSeek-V4.1-Flash	52.7	56.7	74.5	29.3	6.8	70.9	61.4	70.4	40.5	10.1	84.4	**71.9**	**76.4**	34.1	8.3
DeepSeek-V4-Flash	28.8	42.8	65.6	22.7	6.6	51.7	45.2	56.0	42.4	9.2	69.7	56.6	63.1	30.9	12.4
DeepSeek-V4-Pro	45.6	51.1	67.5	23.9	5.0	59.5	50.7	60.7	36.0	5.8	76.2	56.6	63.0	26.5	5.3
Qwen3.5-Plus	38.6	48.6	69.9	25.7	3.9	62.1	53.9	63.8	40.0	7.5	73.1	62.5	68.4	33.7	6.7
Kimi-K3	38.5	42.7	60.1	25.8	4.1	63.9	49.2	60.3	38.4	5.5	81.3	63.7	68.5	31.4	8.4
GLM-5.2	34.3	46.4	69.0	24.2	5.3	57.0	51.4	64.2	36.8	7.9	73.2	59.0	66.6	27.4	7.3
MiniMax-M3	40.3	41.4	62.2	22.0	4.3	59.8	40.9	52.1	33.9	5.7	75.7	52.0	58.1	23.5	4.7
Request-only coder (DC)	–	20.5	–	–	–	–	24.8	–	–	–	–	29.4	–	–	–

Table A10: **Repository track by request view.** All five metrics at each information level (L0 sparse request, L1 named dimensions, L2 qualitative direction); 100 tasks per view, scored on the same frozen targets; RIR uses only eligible episodes. IR, DC, RIR, INT-F1 in %, RG in pp; the final row gives the request-only coder DC on the identical views. Best per column **bold**.

	L0 (sparse request)					L1 (named dimensions)					L2 (qualitative direction)				
Producer	IR	DC	RIR	F1	RG	IR	DC	RIR	F1	RG	IR	DC	RIR	F1	RG
Claude-Fable-5	45.7	44.8	50.5	**25.8**	7.3	**64.2**	**49.9**	51.2	**36.0**	7.5	**83.5**	**57.4**	**53.8**	24.3	10.4
Claude-Opus-4.8	32.8	40.1	46.3	19.6	5.7	56.2	46.1	48.4	33.2	6.0	80.6	54.5	50.9	22.6	13.1
Gemini-3.8-Flash	37.2	**47.2**	**55.3**	19.1	6.4	47.5	47.9	**53.7**	26.0	7.7	78.1	56.3	53.0	20.1	12.7
Gemini-3.5-Flash	14.3	40.3	46.7	11.1	**13.6**	22.1	44.6	42.6	16.6	12.5	48.5	48.4	43.5	13.9	**19.4**
GPT-5.6-Terra	44.7	41.8	45.5	23.2	8.9	59.0	44.4	47.6	28.7	12.1	83.3	52.2	48.6	21.6	14.3
GPT-5.5	**45.8**	43.1	47.6	22.5	8.2	60.2	44.9	47.0	29.6	**13.3**	83.3	54.6	50.9	22.8	12.5
DeepSeek-V4.1-Flash	44.8	44.1	50.2	21.5	6.5	61.1	45.6	47.6	34.4	10.0	82.2	56.5	52.7	**25.9**	10.5
DeepSeek-V4-Flash	25.6	39.4	44.3	19.2	7.1	41.9	40.3	43.0	33.0	9.2	73.9	50.2	46.8	19.5	12.1
DeepSeek-V4-Pro	39.5	43.2	46.6	19.9	8.5	56.9	46.1	47.3	29.8	8.9	78.1	49.6	45.1	22.0	9.6
Qwen3.5-Plus	27.8	43.3	52.0	20.4	7.3	46.2	44.2	47.7	29.2	7.2	68.7	50.6	48.0	20.4	12.1
Kimi-K3	34.7	41.1	46.9	19.3	8.5	56.6	42.8	42.6	31.2	11.8	79.9	51.1	46.6	19.9	10.6
GLM-5.2	28.5	42.2	48.5	18.9	4.2	51.1	45.9	48.3	29.7	6.4	76.0	51.3	47.8	15.9	11.8
MiniMax-M3	31.7	36.4	39.6	17.9	5.1	50.5	34.0	34.2	26.0	12.8	78.2	45.1	40.3	17.6	11.7
Request-only coder (DC)	–	26.7	–	–	–	–	29.3	–	–	–	–	29.3	–	–	–

target set. The pattern alone does not determine whether L2 changes question volume, target choice, or matching quality. Per-view RIR uses a changing recovered set and excludes episodes with no recovered hidden requirements, following Appendix E.1.

I Full thirteen-Producer level-response heatmap

Figure A1 extends the main heatmap to the full roster using identical paired-task calculations. IR and DC use all 300 pairs. RIR uses 121–300 pairs per cell. Across Producers, IR changes range from $+8.1$ to $+24.6$ pp at L1 and $+29.1$ to $+43.6$ pp at L2. The respective DC ranges are -1.1 to $+6.1$ and $+5.8$ to $+17.3$ pp. Each Producer's IR increase exceeds its DC increase in both contrasts, demonstrating that the main figure's distinction between recovery and delivery extends across the roster.

RIR changes range from -8.7 to $+5.8$ pp. Since the recovered set and paired eligibility can change with the view, these values describe conditional realization rather than the fate of a fixed set of newly recovered requirements. The complete grid supports comparing the magnitude and direction of each response without selecting only the strongest models.

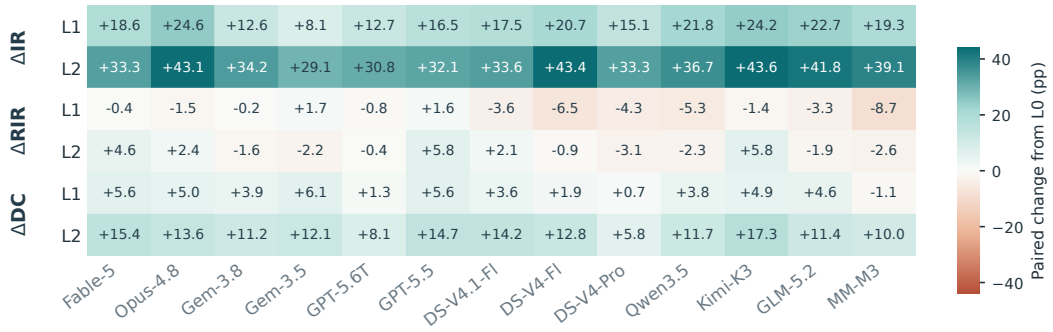


Figure A1: **Paired changes from L0 for all thirteen Producers** (pp). Rows group L1–L0 and L2–L0 contrasts for IR, RIR, and DC. Columns are Producers. IR/DC use 300 task pairs and RIR uses eligible pairs. Values for the seven models in Figure 4 are identical.

J Recovery channels and interaction burden

J.1 Channel realization and route diagnostics

Table A11 gives the values behind Figure 5. For each channel, realization first averages its hidden requirements within a task, then averages tasks supporting that channel. Profile/history has the highest point estimate for twelve of thirteen Producers, and defaults the lowest for nine. These are within-model orderings of means. The displayed intervals quantify uncertainty in each mean.

Route-consistent realization uses the question-selected subset within each task: a validated clarification match is required for clarification targets, whereas profile/default targets must have no matched question identifier. Tasks with an empty route subset are omitted from that route mean. Accordingly, a channel mean and its route mean can differ in both requirements and task support. The clarification route means exceed the channel means by 11–20 pp. This association identifies productive traces linking questions to delivery without assigning a causal effect to asking.

Table A11: **Channel realization at L0** (numeric form of Figure 5). Mean realization (%) of hidden requirements per assigned channel with 95% task-bootstrap intervals, and the route-consistent subset (RC): clarification targets with a validated matched question; profile/default targets without one.

	Clarification		Profile/history		Default	
Producer	mean [95% CI]	RC	mean [95% CI]	RC	mean [95% CI]	RC
Claude-Fable-5	48.1 [45.1, 51.1]	62.2	51.5 [46.5, 56.5]	56.6	47.2 [43.8, 50.5]	45.4
Claude-Opus-4.8	39.7 [36.7, 42.8]	54.9	48.0 [43.0, 53.1]	44.4	39.0 [35.7, 42.4]	37.9
Gemini-3.8-Flash	51.8 [48.8, 54.9]	69.4	58.1 [53.1, 63.2]	63.0	47.4 [44.3, 50.6]	46.0
Gemini-3.5-Flash	35.7 [32.7, 39.0]	53.8	45.0 [39.8, 50.1]	46.5	37.2 [34.0, 40.5]	36.4
GPT-5.6-Terra	48.5 [45.4, 51.7]	59.5	48.2 [43.1, 53.4]	50.0	45.0 [41.5, 48.5]	44.7
GPT-5.5	45.5 [42.4, 48.7]	58.8	49.5 [44.4, 54.5]	46.8	42.7 [39.5, 45.9]	41.1
DeepSeek-V4.1-Flash	48.9 [46.0, 51.8]	62.6	59.4 [54.3, 64.2]	59.3	47.3 [44.0, 50.6]	46.2
DeepSeek-V4-Flash	37.2 [34.3, 40.1]	52.7	41.5 [36.5, 46.4]	40.5	37.2 [34.0, 40.5]	35.5
DeepSeek-V4-Pro	45.1 [41.9, 48.3]	59.4	47.4 [42.1, 52.7]	52.6	43.3 [39.9, 46.8]	42.4
Qwen3.5-Plus	42.4 [39.3, 45.5]	62.1	51.8 [46.7, 56.8]	51.9	42.6 [39.3, 46.1]	40.3
Kimi-K3	37.4 [34.3, 40.5]	53.8	41.3 [36.4, 46.2]	41.9	38.6 [35.3, 41.8]	37.9
GLM-5.2	40.4 [37.5, 43.5]	59.8	47.8 [42.6, 53.0]	48.7	39.9 [36.7, 43.1]	38.2
MiniMax-M3	35.4 [32.5, 38.3]	54.5	38.7 [33.9, 43.6]	40.5	35.2 [32.0, 38.5]	34.1

J.2 Burden and yield by track

Table A12 decomposes the pooled L0 burden summary. Q and Tok. are per-episode means. Tok. estimates question-plus-reply length using the runtime character heuristic. Hit/Q and Hit/k divide pooled distinct clarification hits by pooled questions and by thousands of estimated dialogue tokens. Repeated and unmatched shares also use pooled question counts. A repeated off-route match can contribute to both categories in the main table, so the shares are not a partition.

Table A12: **Burden and yield at L0 by track** (per-track form of the pooled summary). Q: questions per episode; Tok.: estimated question-plus-reply tokens per episode; Hit/Q and Hit/k: distinct validated clarification targets per question and per 1,000 tokens; Rep./Unm.: repeated and unmatched question shares (%).

	Single-file (L0, 200 episodes)						Repository (L0, 100 episodes)					
Producer	Q	Tok.	Hit/Q	Hit/k	Rep.	Unm.	Q	Tok.	Hit/Q	Hit/k	Rep.	Unm.
Claude-Fable-5	9.0	1,598	0.40	2.24	13.7	28.3	8.7	1,590	0.36	1.97	10.9	29.5
Claude-Opus-4.8	6.8	1,489	0.36	1.67	19.7	24.6	6.7	1,498	0.33	1.46	16.8	29.7
Gemini-3.8-Flash	6.2	928	0.40	2.66	9.5	33.5	6.1	937	0.33	2.17	8.3	37.7
Gemini-3.5-Flash	3.2	1,166	0.41	1.15	10.5	31.5	3.2	1,260	0.34	0.86	10.4	38.9
GPT-5.6-Terra	8.9	1,458	0.33	2.01	19.4	31.7	9.2	1,491	0.31	1.92	18.8	29.3
GPT-5.5	9.3	1,293	0.35	2.49	16.3	31.0	8.8	1,279	0.31	2.16	13.0	32.3
DeepSeek-V4.1-Flash	8.9	2,707	0.35	1.13	17.9	32.8	9.2	2,902	0.29	0.93	15.9	34.2
DeepSeek-V4-Flash	6.9	1,385	0.31	1.56	19.7	32.2	7.8	1,648	0.29	1.40	21.0	27.8
DeepSeek-V4-Pro	7.6	2,282	0.32	1.07	18.4	32.8	9.8	2,245	0.26	1.15	20.9	33.8
Qwen3.5-Plus	7.8	1,694	0.34	1.57	19.7	30.3	7.3	1,613	0.32	1.44	14.6	34.6
Kimi-K3	5.8	1,236	0.39	1.83	13.5	27.0	5.8	1,234	0.36	1.67	12.6	27.6
GLM-5.2	6.1	1,225	0.36	1.80	13.2	33.4	6.0	1,235	0.33	1.61	14.0	32.1
MiniMax-M3	5.7	2,226	0.35	0.90	13.6	35.8	5.5	2,131	0.34	0.89	12.1	37.0

Analytic budget ceiling. With at most 14 questions and $c > 0$ fixed clarification targets, perfect distinct matching gives

$$F_{\max}(c) = \frac{2\min(14, c)}{\min(14, c) + c}.$$

The maximum is attained by asking $\min(14, c)$ questions, rather than always using the full quota. The target count exceeds 14 on 41/200 single-file and 47/100 repository tasks, giving task-mean ceilings of 98.6% and 90.3%. This separates the available budget from the Producer's use of it and supplies a common reference for studying targeted clarification.

K Revision dynamics and version selection

K.1 Revision-depth profiles

Table A13 describes how Producers allocate review. Claude-Fable-5, Kimi-K3, Gemini-3.8-Flash, and Claude-Opus-4.8 deliver without revision in 54–62% of episodes, with mean depths of 0.63–0.87. GPT-5.6-Terra, DeepSeek-V4-Pro, and Gemini-3.5-Flash use about 2.0–2.2 revisions on average and reach the four-round limit in roughly a third of episodes. These profiles complement the outcome metrics: models can achieve similar mean delivery through different amounts of review. Zero revisions records the artifact trajectory, not necessarily an explicit acceptance action.

Table A13: **Revision-depth distribution per Producer** (all 900 episodes each: 300 tasks × 3 views). Share of episodes ending at each final revision depth and the resulting mean depth; ReviewGain pools both tracks 2:1 as in Table 2.

Producer	0 rev. (%)	1 (%)	2 (%)	3 (%)	4 (%)	mean depth
Claude-Fable-5	61.4	24.8	7.3	2.7	3.8	0.63
Claude-Opus-4.8	54.0	25.3	8.4	3.8	8.4	0.87
Gemini-3.8-Flash	58.0	28.3	7.0	1.4	5.2	0.68
Gemini-3.5-Flash	17.3	29.2	17.3	6.6	29.6	2.02
GPT-5.6-Terra	17.7	21.8	15.2	12.3	33.0	2.21
GPT-5.5	26.0	30.0	13.7	8.3	22.0	1.70
DeepSeek-V4.1-Flash	29.3	33.6	17.1	8.2	11.8	1.40
DeepSeek-V4-Flash	27.6	23.7	16.0	9.6	23.2	1.77
DeepSeek-V4-Pro	17.7	25.6	13.4	10.2	33.1	2.16
Qwen3.5-Plus	31.7	20.9	11.6	5.1	30.8	1.82
Kimi-K3	62.0	21.3	6.7	3.4	6.6	0.71
GLM-5.2	36.2	24.1	11.0	7.3	21.3	1.53
MiniMax-M3	40.1	25.1	8.8	5.2	20.8	1.41

K.2 GAINS WITHIN FIXED-DEPTH COHORTS

For each final depth k, Figure 7a follows the same episodes from their initial artifact to their delivered version. The nonzero cohorts contain 3,003, 1,382, 758, and 2,246 episodes and gain 10.3, 15.5, 17.4, and 14.7 pp in mean DC, respectively. Every nonzero-depth cohort therefore has a positive net gain. Their equally weighted mean gain is 14.5 pp. It is a depth-balanced descriptive summary, distinct from the episode-weighted ReviewGain that includes the 4,311 zero-revision episodes.

Depth is chosen by the Producer and the cohorts differ in initial DC and model composition. Stratification makes those trajectories explicit but does not remove selection effects or isolate the effect of an extra revision. Figure 7b retains all signed changes, including unchanged and regressing versions.

K.3 OPPORTUNITY FOR VERSION SELECTION

Table A14 measures how often the final artifact scores below an earlier version from the same episode. The affected share ranges from 2.6% to 12.3% across Producers. The mean difference in those episodes is 10.8–18.8 pp. This identifies a concrete direction for improving review: retain earlier candidates and develop reliable stopping or selection criteria. The table uses hidden scores retrospectively. Selecting the best recorded version is therefore a hindsight reference, not a selection policy available to the Producer during evaluation.

Table A14: **Hindsight version selection.** Episodes whose delivered artifact scores below the best version recorded during the episode (900 episodes per Producer); "mean lost" averages the foregone DC over affected episodes.

Producer	affected episodes	share (%)	mean lost DC (pp)
Claude-Fable-5	26	2.9	11.9
Claude-Opus-4.8	23	2.6	16.9
Gemini-3.8-Flash	32	3.6	14.1
Gemini-3.5-Flash	88	9.8	12.1
GPT-5.6-Terra	106	11.8	18.8
GPT-5.5	86	9.6	13.9
DeepSeek-V4.1-Flash	47	5.2	13.3
DeepSeek-V4-Flash	96	10.7	13.1
DeepSeek-V4-Pro	111	12.3	12.1
Qwen3.5-Plus	105	11.7	12.1
Kimi-K3	31	3.4	15.2
GLM-5.2	85	9.4	10.8
MiniMax-M3	79	8.8	16.4

L TASK COMPOSITION AND DELIVERY DISTRIBUTIONS

Aggregate means are complemented by analyses of requirement load, task family, threshold attainment, and paired task-level changes. Each view uses the same frozen task targets and recorded trajectories.

L.1 CONSTRAINT-LOAD TIERS

Tables A15 and A16 partition each track at empirical terciles of hidden-requirement count, keeping tied counts together. The resulting single-file groups contain 83/87/30 tasks and repository groups 38/30/32. Mean DC in the highest-count tier is at least as high as in the lowest-count tier at every view on both tracks. For example, L0 single-file DC is 49.7/47.8/50.4% across the three tiers, and repository DC is 39.6/45.2/42.0%. Requirement count therefore measures catalog size rather than a calibrated difficulty scale. These profiles motivate reporting task content, request view, and track alongside count-based strata.

L.2 TASK-FAMILY BREAKDOWN

Table A17 averages all thirteen Producers and three views within each family. Across the five named single-file families, IR ranges from 56.0% to 62.7% and DC from 47.5% to 62.4%. Feature

Table A15: **Delivery by constraint-load tier.** Tasks split per track at the empirical tercile boundaries of their hidden-requirement count (ties kept together); cells average the thirteen Producers. Counts define catalog-size strata, not calibrated difficulty levels.

			IR (%)			DC (%)		
Track	Tier (hidden reqs.)	tasks	L0	L1	L2	L0	L1	L2
Single-file	T1 (15–19)	83	43.4	62.1	75.5	49.7	52.7	61.4
	T2 (20–21)	87	41.3	60.1	76.3	47.8	52.7	61.7
	T3 (22–28)	30	43.1	61.1	78.9	50.4	55.6	65.9
Repository	T1 (24–29)	38	35.0	53.9	77.7	39.6	42.6	49.9
	T2 (30–32)	30	35.9	52.6	82.8	45.2	46.9	55.7
	T3 (33–75)	32	33.7	48.7	69.1	42.0	44.0	51.4

Table A16: **Per-Producer DC by constraint-load tier** (%, three views pooled). Tiers as in Table A15; single-file T1/T2/T3 hold 83/87/30 tasks and repository T1/T2/T3 hold 38/30/32. Best per column **bold**.

	Single-file			Repository		
Producer	T1	T2	T3	T1	T2	T3
Claude-Fable-5	62.3	61.1	**65.7**	**48.4**	52.7	**51.7**
Claude-Opus-4.8	50.3	52.9	57.4	44.8	49.7	46.8
Gemini-3.8-Flash	63.1	**64.3**	63.0	47.4	**56.2**	48.7
Gemini-3.5-Flash	48.0	47.6	52.7	42.5	48.5	43.0
GPT-5.6-Terra	56.8	57.9	59.4	43.5	49.6	45.8
GPT-5.5	59.5	58.5	60.3	46.7	49.0	47.1
DeepSeek-V4.1-Flash	**63.5**	63.4	62.5	47.9	50.0	48.5
DeepSeek-V4-Flash	49.2	46.3	51.0	41.4	47.8	41.3
DeepSeek-V4-Pro	54.9	50.0	55.1	42.7	50.7	46.5
Qwen3.5-Plus	52.4	56.4	57.9	43.5	49.4	45.9
Kimi-K3	52.1	50.8	54.1	44.2	45.5	45.4
GLM-5.2	53.7	49.5	56.5	42.8	49.1	48.3
MiniMax-M3	43.7	44.4	49.0	37.1	42.3	36.6

development, bug fixing, and operations each contain at least thirteen repository tasks and show lower repository than single-file DC. Research shows the reverse ordering (51.4% versus 47.5%) on six repository tasks. The single repository refactoring task is reported with its sample count rather than treated as a representative family estimate. The breakdown makes the benchmark's coverage and the support behind each comparison explicit.

Table A17: **Results by task family.** IR and DC (%) are thirteen-Producer means with the three request views pooled; family labels follow the task metadata behind Figure 2, with "Other" combining the small other, infrastructure, data-pipeline, and performance labels. Families with few repository tasks are descriptive, not stable estimates.

	Single-file			Repository		
Family	tasks	IR	DC	tasks	IR	DC
Feature	113	59.5	54.2	55	53.4	47.6
Bug fix	39	60.4	55.9	13	54.4	48.1
Operations	14	62.3	55.5	15	56.1	42.0
Research	12	56.0	47.5	6	51.0	51.4
Refactoring	16	62.7	62.4	1	45.5	10.3
Other	6	61.1	51.3	10	59.9	42.4

L.3 Delivery threshold profiles

Table A18 reports the fraction of tasks attaining 25/50/75/100% requirement-level DC at L0 and L2, pooling the two tracks. At L2, Claude-Fable-5 and DeepSeek-V4.1-Flash reach at least 75% DC on 47.7% and 48.0% of tasks, respectively, compared with 24.3% for MiniMax-M3. The highest all-requirement attainment rate is 1.7% at L0 and 8.0% at L2. Threshold profiles thus distinguish broad partial delivery from complete realization of the scored catalog. DC of 100% refers to bound requirement tests. It is not a claim about every untested property of the software.

Table A18: **Delivery reliability profile.** Share of episodes (%) whose final artifact reaches at least the given DC threshold, at the sparsest (L0) and richest (L2) view; 300 episodes per Producer per view. DC $= 100\%$ denotes complete realization of the scored requirement catalog.

	L0: DC $\geq$				L2: DC $\geq$			
Producer	25%	50%	75%	100%	25%	50%	75%	100%
Claude-Fable-5	82.7	58.7	19.0	1.0	88.7	78.3	47.7	7.0
Claude-Opus-4.8	77.0	46.3	12.0	0.7	82.3	65.7	35.3	4.3
Gemini-3.8-Flash	88.7	60.3	20.0	1.7	88.0	75.0	46.0	8.0
Gemini-3.5-Flash	72.0	36.0	9.3	0.3	75.7	58.3	30.3	4.7
GPT-5.6-Terra	80.7	58.0	21.0	1.0	81.3	68.3	37.0	4.7
GPT-5.5	80.3	53.7	16.7	0.3	86.0	74.0	43.3	6.3
DeepSeek-V4.1-Flash	86.0	61.0	17.3	0.3	88.7	79.0	48.0	6.7
DeepSeek-V4-Flash	77.0	38.0	8.7	0.0	82.0	60.0	28.7	3.0
DeepSeek-V4-Pro	80.0	49.0	20.3	0.3	79.3	61.7	31.3	3.7
Qwen3.5-Plus	78.7	48.3	14.0	0.3	83.3	68.0	36.3	4.7
Kimi-K3	73.3	38.7	11.0	0.7	81.7	67.0	41.7	5.7
GLM-5.2	79.3	44.0	11.0	0.7	80.3	65.3	31.0	3.3
MiniMax-M3	73.0	34.0	7.3	0.3	72.3	58.3	24.3	2.7

L.4 Consistency of response to added information

For every Producer, more tasks improve than regress from L0 to L2 (Table A19). Improvement occurs on 57.3–75.3% of tasks and regression on 15.0–30.3%. Claude-Fable-5 has the largest improved share (75.3%) and smallest regressed share (15.0%). Conditional mean gains range from 22.5 to 31.7 pp. Conditional mean losses range from 18.6 to 28.9 pp. Together with positive aggregate gains, these paired results make consistency across reformulated requests a measurable objective. Each pair contains one trajectory per view, so the changes combine the request-view response with trajectory variability.

Table A19: **Per-task response to added information** (L2−L0, DC, 300 paired tasks per Producer). Added information helps on average but not uniformly: the mean change among regressing tasks is of the same order as among improving ones.

Producer	improved (%)	unchanged (%)	regressed (%)	mean + (pp)	mean − (pp)
Claude-Fable-5	75.3	9.7	15.0	+24.4	-20.1
Claude-Opus-4.8	66.7	9.3	24.0	+29.0	-23.8
Gemini-3.8-Flash	67.0	12.7	20.3	+24.4	-25.4
Gemini-3.5-Flash	62.3	10.7	27.0	+31.7	-28.6
GPT-5.6-Terra	60.7	14.0	25.3	+22.5	-21.7
GPT-5.5	71.0	10.7	18.3	+25.5	-18.6
DeepSeek-V4.1-Flash	72.0	11.0	17.0	+24.4	-19.9
DeepSeek-V4-Flash	63.3	10.0	26.7	+28.7	-20.1
DeepSeek-V4-Pro	57.3	12.3	30.3	+23.9	-26.0
Qwen3.5-Plus	65.0	10.7	24.3	+26.3	-22.1
Kimi-K3	70.0	11.3	18.7	+31.2	-24.5
GLM-5.2	67.0	9.0	24.0	+24.0	-19.3
MiniMax-M3	63.3	12.0	24.7	+27.0	-28.9

M Full-specification coder diagnostic

The full-specification reference gives the same `Kimi-K2.7-Code` backend the complete frozen specification in one call, with no Producer, interaction, or revision. It complements the request-only reference by separating a complete input specification from the interactive Producer workflow. It is reported as a diagnostic rather than ranked as a Producer.

DC is 81.6% on 199 single-file tasks and 41.8% on all 100 repository tasks. One single-file task, `ts_029`, is excluded after two provider errors and remains recorded with a missing score. Build/import checks pass for 194/199 single-file artifacts and 84/100 repository artifacts. Their conditional DC means are 83.7% and 49.8%, respectively. The unconditional repository score is below twelve of thirteen Producer means, highlighting the value of measuring a complete workflow alongside

one-shot implementation. Because both input information and process differ, this comparison does not attribute the gap to any individual workflow component or define a delivery ceiling.

N PROMPT LIBRARY

The prompt library covers the evaluation roles and the construction pipeline. Displayed runtime texts are the literal role-specific constants in the released source. The shared client appends the same English-only directive to Producer, Code Agent, and simulator calls. The archived `logged_*` files contain that composed form. The release's prompt audit checks these relationships and SHA-256 hashes against the shipped files. This verifies the published prompt texts, without requiring access to full historical transcripts. Construction prompts are reproduced from source, with excerpts explicitly marked. In the archived texts, gold specifications and gold requirement lists denote the model-assisted targets defined in Appendix B.1. Gold answers and the field `ground_truth_answer` denote construction-generated reference answers. These terms do not denote fresh user feedback. Literal prompts retain their original punctuation for reproducibility.

N.1 PRODUCER SYSTEM PROMPT

The Producer prompt contains the role/action interface and the decision rules. It is identical across tracks. Its illustrative phrase "typically ˜8–12 such hidden decision points" is retained verbatim. It is prompt wording rather than a task-specific catalog count. Actual hidden-target counts are 15–28 for single-file and 24–75 for repository tasks.

Producer prompt, part 1: role and action interface (`SYS_PRODUCER`, opening).

```
You are a Producer Agent: half product manager, half senior engineer. A user has
sent you a VAGUE coding request. The full requirement has HIDDEN details. Your job
    is to recover the
user's COMPLETE true intent and get correct code delivered -- while respecting the
    user's time.

You act ONE step at a time by outputting a single JSON action. The environment runs
     it and returns an
observation, then you act again. You have TWO phases:

GATHER (recover intent, then hand off to the Code Agent). Allowed actions:
  {"thought": "...", "action": "read_profile"}                          # rough
    identity: role, seniority, domains, tech stack
  {"thought": "...", "action": "search_history", "query": "..."}        # search
    this user's past-work history for relevant episodes (returns the most relevant
     sections; empty query returns the section index)
  {"thought": "...", "action": "read_history"}                          # dump the
    FULL interaction history (large; prefer targeted search_history)
  {"thought": "...", "action": "ask_user", "question": "..."}           # ask the
    USER one clarifying question (COSTS the user's time)
  {"thought": "...", "action": "finalize", "brief": "<complete spec for the coder
    >", "requirements": ["<one atomic intent>", ...]}

REVIEW (the Code Agent has implemented your brief; decide if it truly satisfies the
     intent). Allowed actions:
  {"thought": "...", "action": "run_code", "calls": [{"args": [...], "kwargs":
    {...}}], "expectation": "what you expect"}
  {"thought": "...", "action": "ask_user", "question": "..."}           # confirm a
     behaviour with the user
  {"thought": "...", "action": "request_revision", "feedback": "precise changes for
     the coder"}
  {"thought": "...", "action": "accept", "reason": "why it satisfies the intent"}
```

Producer prompt, part 2: decision and review rules (`SYS_PRODUCER`, continued).

```
RULES that determine your score:
  - YOUR PRIMARY GOAL IS TO RECOVER THE **COMPLETE** INTENT. These tasks
    deliberately HIDE many details
    (exact output format & keys, every edge case, tie-breaking, error behaviour &
    messages, empty/None
```

```
    handling, field names/types, defaults). There are typically ~8-12 such hidden
    decision points, and
    EACH wrong or missing one fails tests. Be systematic: before finalizing,
    enumerate every dimension
    that could vary, and for EACH one either (a) infer it confidently from the
    fuzzy goal / persona
    history / a strong convention, or (b) ASK the user. Thoroughness is rewarded;
    leaving a needed detail
    unresolved is the #1 way to fail. It is far better to ask one extra necessary
    question than to guess
    wrong -- you have a generous question budget, use it to nail down the specifics
    .
  - Ask GOOD, SPECIFIC questions (one concrete gap each): exact return shape & key
    names, precise
    behaviour on each edge case, error types/messages, ordering/tie-breaking, what
    empty inputs return.
    The user WILL answer specific questions truthfully.
  - Mild discipline (avoid wasted turns, not a reason to stay silent): the
    INTERACTION HISTORY carries
    real signal about the user's preferences (naming, error handling, tie-breaking,
     idempotency, output
    shape) -- MINE it and INFER rather than asking about things it clearly implies;
     and don't ask about
    things already stated in the fuzzy goal or fixed by obvious convention. The
    history recounts concrete
    past episodes ("they insisted X", "they rejected Y") -- read between the lines.
     Asking about a clearly
    persona-inferable or conventional-default detail is a minor waste (the user
    will deflect it), but
    UNDER-recovering is the far bigger failure -- when in doubt about a load-
    bearing detail, ASK.
  - Your `brief` must be a COMPLETE, unambiguous specification (input shape, output
     format & exact keys,
    every edge case, tie-breaking, defaults, error behaviour) so a coder who sees
    ONLY your brief can
    implement it correctly and completely.
  - In REVIEW, be a rigorous QA engineer. Actually verify with run_code on inputs
    that exercise each
    tricky behaviour (edge cases, errors, ordering), reason about correctness, and
    request targeted
    revisions for anything wrong or incomplete. Do NOT rubber-stamp -- accepting
    broken code is a major
    failure. Use your revision budget; only accept once you have genuinely checked
    the risky behaviours.

Output STRICTLY one JSON object for the single next action. No prose outside the
    JSON.
```

N.2 CODE AGENT PROMPTS

The implementation call supplies the brief and contract. Revision additionally supplies the previous artifact and feedback. Implementation and revision prompts differ per track: the single-file variants demand one self-contained module, the repository variants a complete package honoring the declared architecture.

Single-file implementation (`SYS_IMPL`).

```
You are a senior engineer. Implement EXACTLY the specified function per the brief.
Rules: Python 3.10; standard library ONLY; fully deterministic (no network, files,
    wall-clock, or
unseeded randomness). Return complete, runnable code that defines the entry
    function.
Implement precisely what the brief says -- do not invent behaviour it does not
    state, and do not omit
behaviour it does state.
Output the complete module as ONE ```python fenced code block, and NOTHING else (no
     prose, no JSON).
```

Single-file revision (`SYS_REVISE`).

```
You are a senior engineer revising your implementation. You are given the brief,
your PREVIOUS code, and REVIEW FEEDBACK from the producer describing what to change
    . Apply the
feedback precisely and return the complete revised module.
Rules: Python 3.10; standard library ONLY; deterministic.
Output the complete revised module as ONE ```python fenced code block, and NOTHING
    else (no prose, no JSON).
```

Repository implementation (`SYS_IMPL`, repository).

```
You are a senior engineer. Implement the specified task as a MULTI-FILE PYTHON
    PACKAGE
exactly per the brief and the required module architecture.
Rules:
- Build the package `solution/`: an `__init__.py` that re-exports the entry
    callable (so
  `from solution import <entry>` works), PLUS the submodules named in the
    architecture, each owning
  its slice of the computation and importing siblings with RELATIVE imports (`from
    .parsing import ...`).
- Implement precisely what the brief says -- do not invent behaviour it does not
    state, and do not omit
  behaviour it does state. Python 3.10; standard library ONLY; fully deterministic
    (no network, files,
  wall-clock, or unseeded randomness).
- OUTPUT FORMAT (critical): output EACH file as its own fenced ```python block, and
     immediately BEFORE
  each block put a line exactly:  ### FILE: solution/<path>.py
  Emit every file of the package this way (starting with solution/__init__.py).
    Output NOTHING else --
  no prose, no JSON, no summary.
```

Repository revision (`SYS_REVISE`, repository).

```
You are a senior engineer revising your multi-file implementation. You are given
    the brief,
the required module architecture, your PREVIOUS package (all files), and REVIEW
    FEEDBACK describing what
to change. Apply the feedback precisely and return the COMPLETE revised package (
    every file).
Rules: keep the same module architecture + entry; Python 3.10; standard library
    ONLY; deterministic.
OUTPUT FORMAT: output EACH file as its own fenced ```python block, preceded by a
    line exactly:
  ### FILE: solution/<path>.py
Emit every file (starting with solution/__init__.py). Output NOTHING else (no prose
    , no JSON).
```

N.3 User Simulator prompts

Both simulator operations are identical across tracks. Matching maps a question to at most one hidden requirement. Answering follows the matched channel's policy (Appendix C.4).

Question matching (`SYS_MATCH`).

```
You are standing in for a USER during a requirements dialogue with an assistant.
You are given the task's HIDDEN requirements (each with id, recovery_channel, and
    -- for
clarification ones -- the gold answer), and a single QUESTION the assistant asked.
Decide which ONE hidden requirement the question is about (or none), and how to
    respond.

Output strictly one JSON:
{
  "matched_id": "<requirement id, or empty string if none>",
```

```
  "channel": "<that requirement's recovery_channel, or 'none'>",
  "on_topic_answer": "<if the matched requirement is a clarification one, restate
    its gold answer as one natural sentence; otherwise empty>"
}
Match generously by meaning (paraphrases count), but to at most ONE best
    requirement. Output only JSON.
```

Specification-grounded answering (`SYS_SPEC`).

```
You are the USER who requested this coding task; you know its full specification.
    Answer
the assistant's SPECIFIC question truthfully and concisely, as a user would -- give
     concrete data shapes,
field names/values, or the exact behaviour asked about, grounded in the SPEC.
Rules: answer ONLY what was asked, in 1-3 sentences; do NOT volunteer unrelated
    rules or edge cases they
did not ask about; if the question is over-broad (asks you to enumerate all the
    rules / the whole spec /
"everything" / "all edge cases"), REFUSE and tell them to ask about one specific
    point at a time.
Output only the answer text (no JSON, no preamble).
```

N.4 RECOVERY-JUDGE PROMPT

Each judge receives this system prompt followed by one JSON payload per episode containing the episode header with track, Producer, view, and task identifier, the full atomic-requirement list, and the pre-implementation brief, including any runtime fallback, with a closing reminder that positive evidence must be quoted verbatim from the brief. The verbatim-evidence rule makes votes mechanically checkable. Any `covered=true` decision whose evidence string is not a contiguous substring of the brief is invalid.

Recovery judge (binary entailment per requirement; prompt loaded by `scripts/judge_brief.py`).

```
You are a conservative evaluator of software-requirement recovery.
You will receive a gold list of atomic requirements and the finalized brief that a
    Producer gave to a Code Agent. Decide independently for every gold requirement
     whether the brief explicitly specifies the same required behavior.

Rules:
1. Judge the finalized brief only. Do not use generated code, test outcomes,
    general best practices, or what a competent coder might guess.
2. Evaluate every supplied atomic requirement, including both surface and hidden
    requirements.
3. Mark covered=true only when the behavior is explicit or unambiguously entailed
    by the brief.
4. Merely mentioning the same topic, field, module, or broad goal is insufficient.
5. A contradictory, missing, or materially weaker instruction is not covered.
6. For covered=true, evidence must be copied verbatim from the brief. Do not
    paraphrase, summarize, normalize whitespace, add words, or combine non-
    contiguous snippets. If a short exact quote is hard to find, copy a longer
    exact contiguous substring from the brief. For covered=false, evidence must be
     an empty string.
7. Return exactly one decision for every supplied requirement id and no extra ids.

Return one JSON object with this exact schema:
{"decisions":[{"id":"...","covered":true,"evidence":"exact contiguous substring
    copied verbatim from the brief"}]}
```

N.5 CONSTRUCTION-WORKFLOW PROMPTS

The construction stages in Appendix B use the eleven prompt families below, with track-specific variants. The construction service is `DeepSeek-V4-Pro` throughout. Single-file and repository

tracks use parallel prompt families. Where the two differ only in contract language (callable versus package architecture), the single-file text is shown and the repository variant noted.

Persona construction from one user's session corpus (`SYS_PERSONA`).

```
You are a user-research expert building an evaluation persona for ProdBench. Below
    is an aggregated corpus of ONE engineer's past coding conversations (each line:
     [short_session_id|type|domain] one-line summary || detail || profile_signal ||
     tools).

Your job has TWO distinct audiences, and you must keep them SEPARATE:

(A) A PRODUCER-FACING view the tested agent will actually read at eval time. It
    must be a REALISTIC, UNDER-DIGESTED record: a rough profile (who they are) + a
     rich INTERACTION HISTORY (what they did and said). Crucially, the history
    must let a careful reader *infer* this user's engineering preferences, but
    must NEVER state those preferences as a tidy labelled list. The whole point is
     that the tested agent has to MINE preferences from messy history, not read
    them off a summary.

(B) An INTERNAL GOLD view (never shown to the tested agent) used to design and
    grade tasks: structured conventions, a memory fact base, and a catalog of
    latent preferences each tied to a CONCRETE, UNIT-TESTABLE code rule.

Induce ONLY from the corpus; never fabricate. For weakly-evidenced fields give a
    conservative value or leave empty.

Output strictly one JSON (all free-text in English):
{
  "role": "one-line role",
  "seniority": "junior|mid|senior|staff (best guess from the corpus)",
  "domains": ["frequent domain tags"],
  "tech_stack": ["recurring languages/frameworks/tools"],
  "programming_languages": ["languages this user actually codes in"],

  "interaction_history_md": "PRODUCER-FACING. Markdown. 10-18 concrete past
    episodes, each a short paragraph titled with the domain. Each episode narrates
     a real past task and, most importantly, what the user ASKED FOR, INSISTED ON,
     CORRECTED, or REJECTED -- in natural language, with short quote-like phrases
    where the corpus supports them (e.g. 'they pushed back: the order changed
    between runs, sort by created_at then id'). Write it like a colleague
    recounting history, NOT like a spec. HARD RULE: do NOT use abstract preference
     labels or a bullet list of 'preferences'; do NOT write sentences of the form '
    the user prefers X'. The reader must INFER the preference from the concrete
    episode. Cover every behaviour you encode in `implications` below, so each
    latent preference is genuinely recoverable from this history.",

  "coding_conventions": ["INTERNAL GOLD. engineering habits, e.g. 'prefers explicit
     error codes over exceptions'"],
  "communication_style": "INTERNAL GOLD. one sentence",
  "risk_tolerance": "INTERNAL GOLD. one sentence",
  "behavior_preferences_md": "INTERNAL GOLD. narrative behaviour/communication
    style (markdown, 3-6 paragraphs)",
  "memory_facts": [
    {"fact_key":"preferred_tie_breaking","fact_value":"deterministic, registration-
    order","confidence":0.0-1.0,"source_sessions":["short_session_id"]}
  ],
  "implications": [
    {
      "latent_preference": "a specific latent preference (e.g. 'prefers
    deterministic tie-breaking; dislikes randomness')",
      "code_rule_hint": "the CONCRETE, UNIT-TESTABLE code rule this preference
    implies in a NEW single-function task, phrased as a decidable behaviour that
    differs from the naive default (e.g. 'on equal sort keys, break ties by
    ascending insertion index, never arbitrarily')",
      "evidence_sessions": ["short_session_id", "..."],
      "evidence_quote": "<=15-word phrase grounded in the corpus AND reflected in
    interaction_history_md",
      "implies_requirement_kinds": ["tie_breaking","ordering","determinism"]
    }
  ]
```

```
}

Rules for 'implications' (these drive the benchmark's persona channel, so quality
    matters most):
- Provide 5-9 implications. Each MUST carry a 'code_rule_hint' that is a concrete
    decidable rule for a self-contained pure function (ordering, tie-breaking, a
    specific default value, error/empty handling raise-vs-return, rounding mode/
    precision, boundary inclusivity, output format/key names, validation
    strictness, deterministic vs arbitrary choice). REJECT preferences that cannot
     become a unit-testable code rule (e.g. 'keeps documentation in sync', 'uses
    task tracking') -- those belong only in behavior_preferences_md, not in
    implications.
- Every implication's behaviour MUST be recoverable from interaction_history_md
    alone (an agent reading only the rough profile + history could predict the
    rule). If it is not visibly grounded in the history, drop it.
- ENDORSEMENT (required): 'evidence_sessions' MUST list >=2 DISTINCT
    short_session_id values (from the corpus above) where this same preference is
    actually observable. Only cite ids that genuinely show the behaviour. If a
    preference truly appears in only ONE session, DROP the implication -- a one-
    off is not a stable trait, and a persona rule the tested agent must recover
    has to be repeatedly evidenced. Prefer fewer, well-evidenced implications over
     many single-session guesses.
Output only JSON, no markdown code block.
```

Source-session selection (`SYS_SELECT_SINGLE`; the repository variant `SYS_SELECT_REPO` targets multi-file tasks graded by held-out suites).

```
You are selecting conversations to turn into single-file, function-level coding
    tasks for ProdBench. The Producer Agent starts from a vague request + the user'
    s profile/memory, must infer/clarify/recall the full intent, then deliver a
    self-contained module/function graded by I/O tests.

Final task form: a self-contained single-file solution (>=100 nontrivial lines with
     a real algorithm or stateful component), graded deterministically. The
    conversation may be refactored/abstracted into such a problem (then derivation
    =synthesized) or faithfully reproduced (derivation=faithful).

Below are candidate conversations from one user (summary, type, codeable /
    single_file_extractable / difficulty_potential / self_containment scores,
    profile signal). Evaluate each.

Output strictly one JSON:
{
  "candidates": [
    {
      "session_id": "...",
      "io_codeable": 1-5,
      "self_containment": 1-5,
      "difficulty_potential": 1-5,
      "profile_gap_potential": 1-5,
      "recommended_derivation": "faithful|synthesized|reject",
      "suitability": 0.0-1.0,
      "rationale": "one sentence: why selected/rejected; if synthesized, what self-
    contained function you'd abstract"
    }
  ]
}
Prefer synthesized over reject: if there is any algorithmic/data-processing kernel
    (scoring, parsing, validation, scheduling, state transition, matching, etc.)
    that can be abstracted into a self-contained function, choose synthesized.
    reject only for pure chat / pure lookup-Q&A / pure monitoring with no
    abstractable algorithm.
Output only JSON, no markdown code block.
```

Pass one, task design (`SYS_DESIGN`, excerpt; single-file).

```
You are an evaluation-task designer for ProdBench. Design a self-contained, auto-
    gradable single-file coding task that is DRIVEN BY THIS USER'S PERSONA. The
    Producer Agent starts from a vague spec + this user's rough profile +
```

```
    interaction history, and must infer/clarify/recall the full intent, then
    implement it. The whole benchmark hinges on whether the task's hidden rules
    can be recovered from the persona -- so persona-inferable rules are MANDATORY,
     not decorative.

You are given: (a) this user's GOLD persona implications (each a latent preference
    + a concrete `code_rule_hint` + evidence), (b) a de-identified source
    conversation for domain inspiration. Use the conversation for the DOMAIN/
    algorithm kernel, and the persona implications for the HIDDEN rules.

Hard requirements:
- Output a SINGLE pure function `solve(...)` (you choose the real name + signature)
    : given JSON-serializable args -> a deterministic JSON-serializable value. No
    network/files/time/randomness/global state.
- CRITICAL self-containment: the function MUST NOT perform any network/file/time/
    random/IO operation and MUST NOT import requests/socket/urllib/http/httpx/
    aiohttp/unittest.mock or any third-party module. If the source involves an API/
    service/DB/network, abstract the underlying COMPUTATION into a pure function
    and pass any external behaviour in as PLAIN DATA arguments (the external
    system becomes input data; the function only computes).
- COMPLEXITY MANDATE: design the core so a correct reference is naturally 150-280
    lines and can carry ~20 INDEPENDENT behavioural rules (a second design pass
    will add ~10 more behavioural rules on top of yours, so leave room -- rich
    input structure, several output fields, multiple stages/edge cases). Do NOT
    design a task solvable in under 120 lines. Compose MULTIPLE stages inside the
    one function: (a) validate/parse inputs, (b) a non-trivial core algorithm --
    choose one of: interval/resource scheduling, expression/grammar parsing &
    evaluation, graph traversal / topological sort / shortest path, dynamic
    programming, a small state machine or mini-interpreter, constraint solving/
    assignment, (c) apply the hidden rules, (d) format output. If the source
    domain is simple, ADD structure to reach this bar while staying a single self-
    contained pure function.
- derivation=faithful only if the conversation already holds such a self-contained
    kernel; otherwise derivation=synthesized (use its domain/intent as inspiration;
     explain in synthesis_note).
- Target Python 3.10: match-statements and `X | Y` unions are allowed.

PERSONA-DRIVEN DESIGN (the core of this benchmark):
- The task MUST embed >=4 hidden requirements with recovery_channel=
    profile_inference, EACH instantiated from a DIFFERENT provided persona
    implication. Pick implications whose `code_rule_hint` fits the task domain,
    and bake that exact rule into the task. For each such requirement set:
    implication_ref = the implication id; profile_evidence = {fact_key: the
    implication id, fact_value: its latent_preference VERBATIM, source_sessions:
    its evidence_sessions}. Do NOT invent a persona rule that isn't grounded in a
    provided implication.
- A persona rule must be genuinely RECOVERABLE from the user's interaction history
    by a careful reader (a design-time judge verifies this against the history
    ALONE and DEMOTES non-recoverable ones; the task needs >=2 to SURVIVE, so
    producing 4 gives a safety margin). To pass: (1) pick an implication with
    clear, repeated behavioural evidence in the history; (2) bake in the DECISION
    the history implies (e.g. "captures a reversible snapshot before every write"),
     NOT an over-specific exact format/field-name the history cannot pin down; (3)
     build these from 4 DIFFERENT well-evidenced implications so the task still
    carries >=2 after demotions.


[... omitted for space: atomic-requirement decomposition rules (minimal
    independently testable assertions, bound test inputs, surface/hidden split),
    spec-cleanliness rules, and the 28-line JSON output schema (task_id, title,
    full task_spec, fuzzy_spec draft, entry signature, reference solution, hidden
    requirements with id/statement/category/recovery_channel/criticality/
    clarifying_question/ground_truth_answer/persona_implication_id, surface
    requirements, discriminating test inputs). The full prompt ships verbatim with
     the release. ...]
```

Pass two, requirement expansion from the reference implementation (`SYS_EXPAND`).

```
You are extending a coding-benchmark task to have MANY independently-gradable
    hidden requirements. You are given the FULL gold spec, the correct REFERENCE
    solution (the oracle), and the requirements already written. Mine the
```

```
    reference for ADDITIONAL fine-grained BEHAVIOURAL decisions that the fuzzy
    spec would leave ambiguous, and emit each as a new hidden requirement.

Produce 8-12 NEW hidden requirements. Rules:
- recovery_channel is "clarification" (a concrete rule a user could state if asked
    -- give clarifying_question + ground_truth_answer) or "reasonable_default" (
    give default_justification). Do NOT emit profile_inference (persona rules are
    handled elsewhere) and do NOT emit an output_contract/shape requirement (
    already covered).
- Each must be a SHARP behavioural decision where the obvious commonsense default
    DIFFERS from what the reference does, so a solution written only from a vague
    spec would get a DIFFERENT result on at least one input. Good kinds: secondary
     sort keys, tie-breaking, deduplication, filtering thresholds, specific
    default/fallback values, error/empty/None handling (raise vs return), rounding/
    precision, boundary inclusivity, clamping, ordering of nested collections,
    handling of duplicates/conflicts/missing fields.
- Do NOT duplicate or trivially rephrase an existing requirement. Each new
    requirement must govern a DISTINCT slice of the output (or a distinct error
    condition).
- DECOUPLING (critical): if several rules write to the SAME collector list (issues/
    errors/warnings/results), do NOT grade each with equals/set_equals over the
    whole list -- that makes one wrong rule fail all of them. Instead, for each
    such rule: design its test_inputs to trigger PRIMARILY that one rule, and use
    check.kind="subset" over the collector (e.g. `$.issues[].message`) so grading
    only checks THIS rule's contribution is present. Reserve equals/set_equals for
     slices a SINGLE rule fully determines.
- SLICE CHECK (mandatory): each carries check={"kind":..., "paths":[...]} pointing
    at the SMALLEST output slice it decides. Path grammar (use ONLY this -- NO
    filters/predicates/wildcards like `[?x=y]` or `*`): "$"=whole; "a.b"=nested
    key; "a[].b"=field b of each element of list a; "a[2]"=index. For a rule about
     a subset of a list, point at the whole list field and use set_equals/equals;
    do NOT invent a filter. kinds: equals (ordered compare), set_equals (order-
    independent), len_eq (count), key_present, raises (arg=exception type name,
    paths ["$"]), regex_match (arg=regex), numeric_close (arg=tolerance).
- test_inputs: >=2 COMPACT inputs on which a naive rule-ignoring implementation
    differs from the reference ON THIS REQUIREMENT'S SLICE. Keep inputs small.
    Each input dict's keys MUST be EXACTLY the function parameter names in the
    signature (the oracle calls the entry as `fn(**input)`).

Output strictly one JSON:
{"requirements": [
  {"id":"rx1","statement":"...","category":"tie_breaking|default_value|edge_case|
    error_handling|filtering|...","visibility":"hidden","recovery_channel":"
    clarification|reasonable_default","criticality":"must|should","check":{"kind
    ":"...","paths":["..."]},"clarifying_question":"...","ground_truth_answer
    ":"...","default_justification":"...","test_inputs":[{"input":{...},"note":"
    why it discriminates"}]}
]}
Output only JSON, no markdown code block.
```

Naive discriminator solution (`SYS_NAIVE`).

```
You ONLY receive a task's FUZZY spec and function signature (no hidden rules).
    Write an honest but under-informed implementation: implement the stated core
    logic correctly, and for anything the fuzzy spec does not mention (output
    format/ordering/tie-breaks/defaults/boundaries/error handling) use the most
    intuitive commonsense default. Do NOT invent special rules the fuzzy spec
    never stated.
Python 3.10 compatible, standard library only, same entry-function name, no errors
    on normal inputs.
Output strictly one JSON: {"naive_baseline_py": "<complete python>"}. Output only
    JSON.
```

Test augmentation for non-discriminating requirements (`SYS_AUGMENT`).

```
The following HIDDEN requirements have no test input that distinguishes a correct
    solution from a naive one ignoring them. For EACH, add 1-2 inputs on which the
     correct reference and a naive (rule-ignoring) implementation give DIFFERENT
```

```
    results. You are given the full spec, the reference, the naive baseline, and
    the requirements to cover. Provide inputs only (no expected).
Output strictly one JSON: {"new_inputs":[{"req_id":"r3","input":{...},"note":"why
    it discriminates"}]}. input must be JSON-serializable. Output only JSON.
```

L0 view derivation, information removal only (`SYS_FUZZY`).

```
You are given a task's full Gold spec, its function signature, and the list of
    HIDDEN requirements. Produce a genuinely VAGUE fuzzy spec by ONLY REMOVING
    information (fuzzy must be a strict subset of the full spec).

Keep: the high-level goal (1-2 sentences), the function signature, AND the OUTPUT
    CONTRACT -- the exact set of output keys, their value types, and the container/
    nesting shape (the `output_contract` surface requirement). The shape is a
    mechanical API contract, not a hidden intent; stating it lets a solver produce
     the right STRUCTURE and get partial credit on behaviour, so keep it.
REMOVE / do NOT reveal (these are the hidden BEHAVIOURAL rules): output ordering,
    tie-breaking, deduplication, filtering, default values, error/empty-input
    handling, rounding, boundary behaviour, AND any description of the algorithm
    or detailed input-field semantics that would let a reader INFER those hidden
    rules. Do NOT hint that the function corrects/fixes/normalizes/adds/removes/
    falls-back/validates anything; do NOT mention specific entity counts, parent/
    child or type->shape mappings beyond the bare output shape, or which
    transformation is applied -- those are exactly the hidden rules. A competent
    engineer reading the fuzzy spec must know WHAT SHAPE to return but have to
    GUESS every behavioural detail that fills it; if they could deduce a hidden
    behavioural rule from your fuzzy spec, you revealed too much.

Output strictly one JSON (English):
{
  "fuzzy_spec_md": "## Goal (1-2 sentences) / ## Function signature / ## Output
    shape (the exact keys/types/container -- the output contract, stated plainly) /
     ## Rough input-output idea (high level only). NO hidden behavioural rules and
     NO algorithm hints.",
  "initial_request": "the user's vague opening line (colloquial, no implementation
    details)"
}
Output only JSON, no markdown code block.
```

L1/L2 graded-hint generation (`SYS_HINT`, repository variant shown; the single-file variant defines the same soft/strong tiers with a less explicit enumeration of the permitted direction words).

```
You write GRADED, DELIBERATELY VAGUE hints for a coding-benchmark task, to create
    difficulty tiers between a full spec and a fuzzy spec.

You are given: the FULL gold spec, the FUZZY spec (a vague subset the solver
    actually starts from), and a list of HIDDEN requirements that the fuzzy spec
    intentionally omits. For EACH hidden requirement you must produce TWO hints of
     increasing explicitness:

- "soft": ONE sentence. Point ONLY at the aspect/dimension the fuzzy spec leaves
    undecided, and say the solver must decide it. Give NO indication of the
    correct resolution. Good soft hints read like "the spec does not say how X
    should be handled --- decide this deliberately" or a neutral question. A
    reader should know WHERE a decision is needed but get zero help on WHAT to
    decide.

- "strong": ONE sentence, MEANINGFULLY more informative than the soft hint (a
    reader with the strong hint must be clearly better positioned than one with
    only the soft hint --- do NOT merely reword it). COMMIT to the DIRECTION /
    POLARITY / qualitative choice of the decision so a competent reader is nudged
    the right way, e.g. "results are ordered from highest to lowest", "the
    boundary is inclusive", "malformed input raises rather than returning a
    default", "ties keep the earliest item", "counts round down", "empty groups
    are dropped rather than kept". State that qualitative direction plainly. You
    must NOT state the literal, test-checkable TOKENS: NEVER write exact output
    key/field names or exact output shape; exact numeric thresholds, lengths, or
    counts; the exact error type name or message text; or the exact sort KEY /
```

```
    attribute names the rule uses. Direction words (ascending/descending,
    inclusive/exclusive, raise/return, keep-first/keep-last, round up/down, drop/
    keep, before/after) ARE allowed and ENCOURAGED --- a test still cannot be
    passed by copy-paste because the exact key/threshold/type stays hidden and the
     reader must still implement the precise rule with it.

HARD RULE: neither hint may let a reader pass the hidden test by copy-paste. The
    hints ADD NO checkable specification text --- they only guide attention and
    lean. Do not reveal that hidden requirements were "removed"; write the hints
    as natural engineering considerations.

Output strictly one JSON object, English, covering EXACTLY the given requirement
    ids:
{"hints": [{"id": "r6", "soft": "...", "strong": "..."}, ...]}
Output only JSON, no markdown code block.
```

L2 literal-leak audit (`SYS_HINT_LEAK`, repository variant; the single-file audit applies the same literal-token rule).

```
You audit whether STRONG hints for a coding task leaked the literal answer to
    hidden requirements. For each requirement you get its statement (and, when
    present, the ground-truth answer) and the strong hint. A hint is LEAKED ONLY
    IF it hands over the literal, test-checkable TOKENS: the exact output key/
    field names or shape, an exact numeric threshold/length/count, the exact error
     type name or message text, or the exact sort KEY / attribute the rule uses. A
     strong hint that commits to the DIRECTION / POLARITY of a decision (ascending
     vs descending, inclusive vs exclusive, raise vs return, keep-first vs keep-
    last, round up vs down, drop vs keep, before vs after) WITHOUT naming the
    exact key/threshold/count/error-type/field is NOT leaked --- that qualitative
    direction is the intended strength of the L2 tier. Only flag hints that expose
     the exact checkable tokens.
Output strictly one JSON: {"leaked": ["r_id"...], "ok": true|false, "reason": "one
    sentence"}. Output only JSON.
```

L0 leakage audit (`SYS_LEAK`).

```
You audit whether a FUZZY task spec LEAKED hidden requirements. You get the FUZZY
    spec and the HIDDEN requirements (each tagged with recovery_channel); you do
    NOT get the persona.
A hidden requirement is LEAKED only if its answer is stated in or directly
    deducible from the FUZZY TEXT ALONE. NOT leakage: profile_inference (recovered
     from persona you can't see), reasonable_default (guessed by convention unless
     the value is stated in fuzzy), clarification (needs a question unless fuzzy
    already pins it).
Output strictly one JSON: {"leaked": ["r_id"...], "ok": true|false, "reason": "one
    sentence"}. Output only JSON.
```

History-recoverability audit, Producer-facing view only (`SYS_RECOVER`).

```
You are given ONLY what the tested agent will see about a user: a ROUGH public
    profile and the user's raw INTERACTION HISTORY (past episodes). You CANNOT see
     the task's gold spec or any tidy preference list. Below is a hidden
    requirement claimed to be inferable from this user. Decide whether a careful
    agent, reading ONLY this profile + history and MINING it, would arrive at this
     rule.
Answer true if the history contains concrete episode(s) that make this rule a
    natural, well-supported inference for a careful agent -- the exact wording,
    field names or output format need NOT appear verbatim; a clearly-evidenced
    BEHAVIOURAL PATTERN pointing at this rule is enough. Answer false only if the
    history gives no real behavioural support, so adopting the rule would be a
    blind guess.
Output strictly one JSON: {"recoverable": true|false, "reason": "one sentence"}.
    Output only JSON.
```

O RELEASE CONTENTS AND REPRODUCTION

The review package contains three connected layers. **Dataset:** 300 task packages with callable contracts, request views, persona/history, atomic requirements, test bindings, references, and executable tests. **Measurements:** 11,700 episode records, requirement-level votes and execution outcomes, baseline scores, interaction summaries, version-score trajectories, and anonymous human labels. **Implementation:** construction and evaluation workflows, prompt texts, metric computation, and main-text and appendix renderers, together with the synchronized manuscript.

Offline reproduction. After installing the supplied requirements, `bash reproduce.sh` verifies evidence checksums, recomputes metrics, checks recorded outcomes and complete vote panels, and regenerates the main exhibits. Numeric aggregates match frozen outputs to 10^{-9} and generated tables match the manuscript byte for byte. Then run `python reproduction/src/render_appendix.py` to regenerate appendix tables and the full heatmap. The appendix and prompt audits check the displayed claims, generated tables, and source-to-prompt relationships. Full commands and file mappings appear in `docs/REPRODUCIBILITY.md`.

Executable validation and new evaluation. `scripts/validate_reference.py` executes all 300 released references. `scripts/smoke_workflow.py` exercises both workflow paths without model calls. New evaluations generate new artifacts through the same interfaces. Independent recovery judgments can be supplied through `scripts/judge_brief.py`. Recorded measurements also support alternative aggregation, panel thresholds, and interaction analyses without API calls.

Evidence scope. Original source conversations, full Producer transcripts, original brief texts, and all historical generated artifacts are not distributed. The package supports independent recomputation of recorded metrics and human agreement, but not fresh semantic rejudging or execution of every original model artifact. Anonymous labels omit identifying workbook metadata and free-text notes. Code is licensed under MIT and task data under CC BY 4.0.