SWE-PLAN: BENCHMARKING CODING AGENTS BEFORE THEY CODE

Anonymous authors
Paper under double-blind review

ABSTRACT

Before implementation, current coding agents typically clarify requirements, ask for critical decisions, and form a plan, a process that mainstream products have formalized as Plan Mode. However, existing benchmarks largely entangle this pre-implementation phase with code execution and overlook requirement clarification, leaving this phase unanalyzed and the effects of models and harnesses unattributed. To bridge this gap, we introduce **`SWE-Plan`**, the first benchmark that isolates the complete pre-implementation phase and evaluates it directly. Grounded in requirements engineering, **`SWE-Plan`** comprises **392** repository-level tasks with **929** withheld critical decisions and **4,937** atomic planning obligations, assessing whether agents recover these decisions and produce complete, repository-consistent plans, with metric reliability validated by human judgment and downstream execution. Evaluating **8** models under **3** mainstream harnesses, we demonstrate that **(I)** clarification and planning are correlated within a system at $r = \mathbf{0.62}$ yet not interchangeable; **(II)** failures in both dimensions concentrate on requirements and verification obligations that are never explicitly stated; and **(III)** planning rankings correlate at $\rho = \mathbf{0.63}$ across harnesses, while a **5.5** pp change in ask rate accompanies only a **0.8** pp change in question precision, indicating that the base model mainly determines planning and questioning ability, while the harness mainly shifts the willingness to ask. **`SWE-Plan`** provides a diagnostic and systematic evaluation foundation for optimizing coding agents and dedicated planners.

1 INTRODUCTION

In current practice, a coding agent devotes time to thinking and analysis before it modifies a repository. It analyzes the repository and the user's request, asks the user questions, and formulates a plan for implementation. Software engineering terms this phase the *pre-implementation phase* (Riddle & Edwards, 1980), divides it into requirements and design as activities distinct from implementation (Washizaki, 2024), and widely recognizes its importance (Brooks, 1987). Industry products have equipped this phase with a dedicated interface. Mainstream harnesses, including Codex, Claude Code, Cursor, Copilot, and OpenCode, all provide a dedicated Plan Mode, in which the agent explores the repository in read-only mode, asks the user to clarify critical decisions, and delivers a reviewable plan before implementation begins (OpenAI, 2026a; Anthropic, 2026a; Cursor, 2025; GitHub, 2026; OpenCode, n.d.). Practice and theory therefore point to the same conclusion. The ***pre-implementation capability*** is distinct from subsequent execution and constitutes an important component of coding agent capability (Jiang et al., 2024).

However, existing evaluations have not yet assessed this capability. SWE-bench and its successors judge whether a final patch passes tests (Jiang et al., 2025). Although the quality of the pre-implementation phase does influence final performance, it is entangled with the other capabilities of the model and cannot be evaluated on its own. Clarification fares even worse, because the requirements are fully contained in the issue, leaving the agent neither the need nor the opportunity to ask (Jiang et al., 2026). Other benchmarks move upstream but capture only fragments of this phase, evaluating in isolation either questions under ambiguity (Li et al., 2023; Vijayvargiya et al., 2026; Fang et al., 2026) or plans and designs produced without interaction (Bairi et al., 2024; Hamblin et al., 2026). As a result, existing evaluations can tell neither whether a failure originates before or during execution nor whether an improvement comes from the model or the harness. Moreover, planning is increasingly delegated to dedicated components, from standalone planners (Erdogan et al., 2025;

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.

Phan et al., 2024) to orchestrators coordinating tasks in multi-agent systems (Fourney et al., 2024; Anthropic, 2025; Yue et al., 2025; Su et al., 2025), yet the signals used to evaluate them still come only from end-to-end outcomes, leaving a blind spot in their optimization.

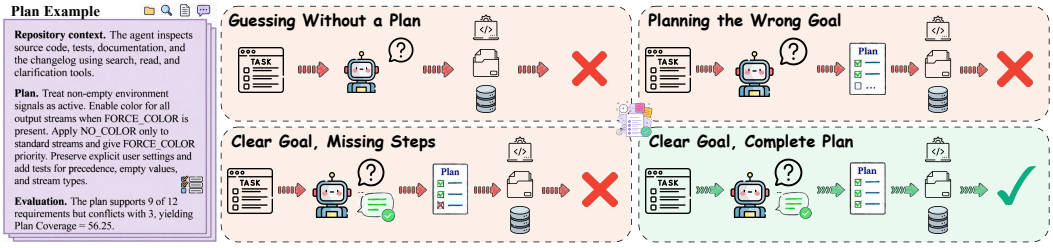


Figure 1: Clarification and planning do not imply each other. A plan can be detailed yet rest on a guessed requirement, and fully correct clarification can still yield a plan that omits essential steps.

To address this gap, we introduce **SWE-Plan**, the first benchmark that isolates the complete pre-implementation phase from execution and evaluates it comprehensively. Grounded in requirements engineering (Zave & Jackson, 1997), we derive two dimensions of pre-implementation capability, namely *clarification* and *planning*, as illustrated in Figure 1. **SWE-Plan** adopts the read-only Plan Mode as a standard interface, so that the two dimensions are measured separately within the same interaction under ecologically valid conditions. We construct **392** tasks from **154** real repositories. Through manual revision, each task withholds 2–4 decisions that materially change the implementation strategy, **929** in total. Clarification is evaluated by a human-calibrated, memory-augmented User Simulator, which matches each question from agents against the withheld decisions and returns the corresponding answers. The plan produced after clarification is then judged by a memory-augmented agentic Plan Judge against task-specific obligation checklists containing **4,937** planning obligations. Meta-evaluation shows that the two evaluators reach Cohen's κ values of **0.89** and **0.76** with human experts, and that higher plan scores correspond to higher downstream execution success, validating the reliability and validity of our metrics.

Evaluating **8** models under the native Plan Mode of **3** harnesses, we show that **SWE-Plan** clearly separates agent configurations and captures planning and clarification signals that outcome-level evaluation confounds or never elicits. We further find that **(I)** clarification and planning are coupled within a system, as their changes under a fixed model and task correlate at $r = \mathbf{0.62}$, yet are not interchangeable across models, **(II)** failures in both dimensions tend to arise where no input states the requirement explicitly, from withheld decisions in clarification to unrequested verification in planning, and **(III)** planning rankings stay consistent across harnesses at $\rho = \mathbf{0.63}$, while harness changes shift the ask rate by **5.5** points but question precision by only **0.8** points, linking planning and question precision to the base model and willingness to ask to the harness. Our contributions are summarized as follows:

❶ ***Theory-Grounded Benchmark.*** We introduce **SWE-Plan**, the first benchmark that isolates the complete pre-implementation phase from execution. Grounded in requirements engineering, it derives clarification and planning as two dimensions, instantiates them in **392** repository-level tasks, and evaluates them with ecological validity through the Plan Mode interface, offering stable evaluation of agents and orchestrators.

❷ ***Execution-Decoupled Evaluation.*** We design dedicated metrics for clarification and planning decoupled from code execution, scored by memory-augmented evaluators. The metrics align closely with human experts and predict downstream execution, showing that pre-implementation capability can be measured reliably on its own.

❸ ***Validation & Insights.*** Extensive experiments validate **SWE-Plan** and reveal a shared weakness of current agents in implicit requirements and the distinct roles of models and harnesses in this phase. These insights call for optimizing the pre-implementation phase as a whole, guiding the development of future models.

2 Related Work

Coding Agent Benchmarks. Repository-level coding benchmarks exemplified by SWE-bench evaluate whether a final patch passes executable tests (Jimenez et al., 2024). SWE-bench Verified,

SWE-bench Multimodal, and SWE-bench Pro improve task validation or coverage without changing this endpoint (Chowdhury et al., 2024; Yang et al., 2025; Deng et al., 2025). The protocol establishes functional outcomes but folds requirement interpretation, repository analysis, planning, implementation, and debugging into one output, and a recent survey likewise finds that benchmarks targeting requirements engineering and planning remain scarce (Wang et al., 2025). Work that moves upstream remains divided by capability. Planning-oriented coding benchmarks assess design documents produced from specifications or intermediate reasoning against reference trajectories, but use fully specified inputs, so the agent never needs to elicit what the request leaves open (Li et al., 2024; Liu et al., 2026). Clarification benchmarks instead study question asking, dialogue, and restraint under underspecified instructions (Mu et al., 2024; King & Flanigan, 2026; Ji et al., 2026). HiL-Bench injects human-validated blockers into repository tasks, matches each question against blockers, and scores selective escalation (Trinh et al., 2026), but its tasks still end in execution and the plans with recovered information are never evaluated. SWE-RPG evaluates the complete workflow of coding agents, but it relies on patch resolution rates to analyze clarification and planning retrospectively and thus still falls short of a complete analysis of this phase (Zhou et al., 2026). These designs provide crucial groundwork but fail to evaluate the complete pre-implementation phase directly and jointly.

Pre-Implementation Theory. Software engineering has long studied the requirements and design preceding implementation as an independent phase, with dedicated theories and evaluation methods (Royce, 1970; ISO, 2017). Defects introduced in this phase become more costly to fix the later they are found (Boehm, 1981; Boehm & Basili, 2001). Prior research notes that even when development iterates between phases, pre-implementation artifacts should still be produced, since developers rely on them for overall review and reduced debugging effort (Boehm, 1988; Parnas & Clements, 1986). On the requirements side, requirements engineering decomposes the requirements activity into elicitation, specification, and validation, and defines elicitation as the proactive identification of stakeholder needs (Christel & Kang, 1992; ISO, 2018; Nuseibeh & Easterbrook, 2000), which grounds clarification. On the specification and design side, Zave & Jackson (1997) characterize an adequate specification as one that, together with domain knowledge, entails all requirements, which grounds planning. For evaluation, software inspections and active design reviews evaluate design artifacts before coding through explicit review questions (Fagan, 1976; Parnas & Weiss, 1985; Basili et al., 1996), making the evaluation of pre-implementation artifacts a mature practice in software engineering. Nevertheless, coding agent evaluation has not yet caught up, failing to translate this phase separation and practice into measurement.

3 SWE-PLAN: BENCHMARK DESIGN

SWE-Plan turns the pre-implementation phase into a measurable object, as shown in Figure 2. Section 3.1 formalizes the phase and its two dimensions. Section 3.2 describes how tasks are built from real repositories, how plan-critical decisions are withheld, and how tasks are accepted.

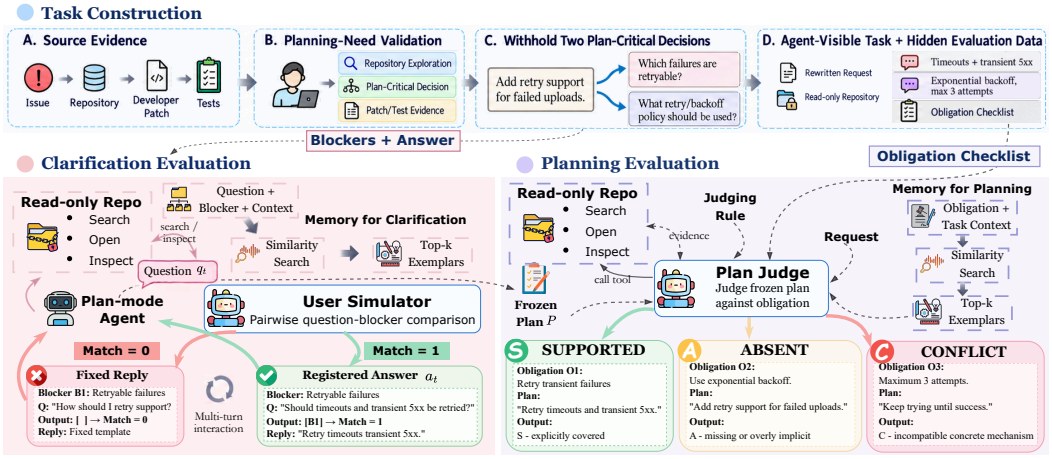


Figure 2: Overview of **SWE-Plan**.

3.1 Problem Formulation

We formalize the pre-implementation phase and operationalize it through Plan Mode, the interface that existing products provide for this phase. This operationalization has both ecological and theoretical validity, as mainstream harnesses have converged on Plan Mode and its boundary coincides with the phase separation of software engineering. We then decompose the capability exercised in this phase through requirements engineering.

Pre-implementation phase. In Plan Mode, an agent receives a code repository $\mathcal{C}$ and a software request x from the user. It retrieves evidence from $\mathcal{C}$ without modifying existing files, asks the user questions q_t about information that the repository cannot provide, and incorporates each reply into the same interaction until it returns a plan P. We formalize this process as

$$(\mathcal{C}, x) \mapsto \big[\, e_t,\ q_t,\ a_t = \mathrm{User}(q_t) \,\big]_{t=1}^{T} \mapsto \big(q_{1:T},\ P\big), \qquad \Delta\mathcal{C} = \varnothing, \tag{1}$$

where each round t retrieves evidence e_t from $\mathcal{C}$, asks any number of questions q_t, and receives their replies a_t, each conditioned on all previous rounds. The constraint $\Delta\mathcal{C} = \varnothing$ marks the boundary of the phase, i.e., the process ends with a reviewable plan before any change to the repository.

Planning and Clarification. Zave & Jackson (1997) characterize a correct specification S as one that, together with domain knowledge K, entails the requirements R, written $S, K \vdash R$. In our setting, R is the set of requirements that the intended change must satisfy, $K_{\mathcal{C}}$, the knowledge that holds in $\mathcal{C}$, plays the role of K, and the plan plays the role of the specification handed to implementers, so a plan is adequate when $P, K_{\mathcal{C}} \vdash R$. The request covers only part of R, and we define the *blockers* of a task as the requirements that neither the request nor the repository settles. The resulting correspondence is

$$S, K \vdash R \ \ \rightsquigarrow \ \ \mathcal{B} = \{\, r \in R : x, K_{\mathcal{C}} \nvdash r \,\}, \qquad \underbrace{x, K_{\mathcal{C}}, a_{1:T} \vdash R}_{\text{clarification}}, \qquad \underbrace{P, K_{\mathcal{C}} \vdash R}_{\text{planning}}. \tag{2}$$

This view yields the two dimensions of pre-implementation capability, one for each output of Eq. (1), with repository exploration serving both. *Clarification* recovers $\mathcal{B}$ from the user, obtaining answers $a_{1:T}$. *Planning* produces P with $P, K_{\mathcal{C}} \vdash R$.

Properties of the formulation. Built on Eq. (1) and (2), **`SWE-Plan`** has three properties that make the phase measurable on its own. ❶ Since $\Delta\mathcal{C} = \varnothing$, every scored output precedes any change to the codebase, so both scores are attributable to the pre-implementation phase. ❷ Every task is constructed with $\mathcal{B} \neq \varnothing$, so the request and repository alone never determine an adequate plan, and clarification becomes necessary. ❸ $q_{1:T}$ and P come from the same interaction, so the two dimensions, scored as $\mathrm{Clar}(q_{1:T})$ and $\mathrm{Cov}(P)$, are measured jointly and their coupling becomes observable.

3.2 Task Construction

Repositories and Tasks. **`SWE-Plan`** is built on 154 open-source repositories. Starting from candidates that were initially constructed with SWE-Factory (Guo et al., 2026), we manually retain maintained projects with active continuous integration and reproducible builds, exclude AI-dominated projects, and strip instruction files intended for coding agents. Each repository is pinned at a verified revision in a dedicated container image with its dependencies preinstalled. For each repository, SWE-Factory yields SWE-style base tasks, each comprising an issue-style description, a developer patch, and executable tests. Two experts then retain only tasks that require planning to be solved effectively, i.e., tasks requiring repository exploration, coordination of multiple behavioral or compatibility requirements, and implementation decisions with more than one plausible resolution. To reduce data contamination, every repository is pinned at a commit later than June 1, 2026, after all published training cutoffs of the evaluated models, and all rewritten requests, blocker registries, and obligation checklists are newly constructed and unreleased. The screening criteria are detailed in Appendix C.

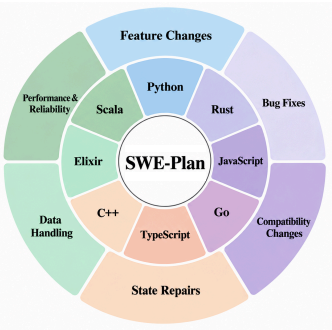


Figure 3: Statistics of **`SWE-Plan`** are detailed in Appendix B.

Blocker Annotation. These tasks are manually annotated on the current repository state, adopting the notion of blockers from HiL-Bench. Instead of injecting blockers as that work does, we compare the issue, the base repository, the developer patch, and the tests to recover decisions that the developer patch and tests have already settled. Following Eq. (2), a decision is retained as a blocker only when the repository cannot determine its value and at least two plausible answers would lead to materially different correct plans. Appendix D details the blocker criteria. Each blocker is recorded in a *blocker registry* with the unresolved decision, the registered answer, its effect on the plan, and supporting patch or test evidence. We then remove each registered answer from the request and, if necessary, from the container image, but keep the symptoms and scope of the task, so each blocker can still be noticed during exploration, but its answer cannot be inferred. The rewritten request therefore remains understandable on its own but determines no single plan without the withheld information.

Acceptance Checks. A task enters **`SWE-Plan`** only if it passes two groups of checks: ❶ ***Executability.*** The edited container must build, and the evaluation tests must fail on it and pass after the developer patch is applied. ❷ ***Planning Necessity.*** Two experts independently check every task. They verify that the task requires planning and that every blocker meets all admission criteria above. They further verify that the rewritten request is understandable on its own, that neither it nor the container reveals any registered answer, and that each registered answer changes the plan as recorded. Tasks failing any check are revised and reviewed again. All 392 tasks across 6 types of tasks and 8 programming languages as shown in Figure 3, with 929 blockers in total, pass both groups of checks. The acceptance procedure is detailed in Appendix E, and Appendix F gives complete examples.

4 Evaluation

`SWE-Plan` extends an established protocol for clarification and introduces a new protocol grounded in software engineering for planning. Clarification evaluation follows HiL-Bench (Trinh et al., 2026) in matching every question against registered blockers and summarizing the matches by Ask-F1, and embeds this matching in the same Plan Mode session so that the matched answers feed the plan under evaluation. Planning evaluation is the core contribution of **`SWE-Plan`**. It turns the adequacy condition $P, K_{\mathcal{C}} \vdash R$ into checkable planning obligations in the manner of proof obligations in refinement (Abrial, 1996), organizes them as an inspection checklist (Fagan, 1976), and formulates every obligation as singular and verifiable, following ISO/IEC/IEEE 29148 (ISO, 2018).

Both dimensions are scored by two judges that share a memory-augmented agentic design (Luo et al., 2025). A judge is a read-only coding agent adapted from mini-SWE-agent (Yang et al., 2024) that inspects the repository at its pinned revision and returns a separate discrete judgment for every output–reference pair, from which all scores are computed deterministically. Following this paradigm, the judge is calibrated by a frozen memory of 40 manually curated cases and retrieves the 5 cases most similar to the current comparison as precedents. Both memories follow the same annotation principle, in which two experts label every case independently with a rationale for the boundary and adjudicate disagreements to consensus, as detailed in Appendix G.

4.1 Clarification Evaluation

User Simulator. The User Simulator is in effect the clarification judge, acting inside the interaction. It reads every question from the agent and judges it against every blocker registered for the task. A question–blocker pair is matched, coded as a binary 1, when a direct answer to the complete question would determine the decision that the blocker represents, given the request, the evidence gathered so far, and the preceding interaction, and one question may match several blockers. Topical mentions, generic requests for detail, repeats of matched decisions, and questions that the repository can answer are not matched and receive a fixed template reply. For every newly matched blocker, the User Simulator returns the registered answer, and the session continues. Its judgments are final for scoring and determine the user's replies. Appendix H gives its full specification.

Metrics. From these binary matches, a question is *productive* if it matches at least one blocker that no earlier question has matched, and $\mathcal{B}^\star \subseteq \mathcal{B}$ denotes the blockers matched in a session. For a session with n_{q} questions, of which n_{prod} are productive, $\mathrm{Clar}(q_{1:T})$ is measured by Productive Precision $\mathrm{Prec} = n_{\mathrm{prod}}/n_{\mathrm{q}}$ and Blocker Recall $\mathrm{Rec} = |\mathcal{B}^\star|/|\mathcal{B}|$, with Prec averaged only over sessions that

ask. We summarize the two by their harmonic mean, Ask-F1. These metrics measure how accurately an agent asks. Because the same Ask-F1 can arise from silence or from excessive unproductive questioning, we separately measure the willingness to ask by the Ask Rate, the fraction of sessions with at least one question, and Questions per Session, the mean of n_{q}.

4.2 Planning Evaluation

Obligation. Software tasks admit multiple correct implementations, so comparison with a single reference plan would reward one route rather than planning quality. Following software inspection practice, in which explicit review questions focus attention on required design properties (Parnas & Weiss, 1985; Basili et al., 1996), we instead specify what any adequate plan must address. A *planning obligation* is a property that a plan must address for $P, K_{\mathcal{C}} \vdash R$ to hold and to be checkable, and the obligations of a task form its *obligation checklist* $\mathcal{O}$. Obligations take five roles: *behavior* and *condition*, derived from R, and three plan-level duties, namely *ownership* of where the change belongs (Parnas, 1972), *preservation* of what must remain unchanged (Borgida et al., 1995), and *verification* of the result. Checklists are constructed and frozen before any candidate plan is evaluated. Coding agents propose and analyze diverse implementation routes from the request, repository evidence, resolved requirements, developer patch, and tests. Human experts compare these routes against the source evidence, reject infeasible alternatives, merge equivalent choices, and remove cosmetic details and structures specific to the developer patch. The shared correctness properties become obligations, each atomic in stating exactly one property that can be judged on its own and paired with a near miss, a plausible mechanism known to violate it, while valid alternative routes are recorded for the whole checklist. Appendix I details the design of the five roles and the construction procedure.

Plan Judge. The Plan Judge acts after the plan is frozen. Its read-only sandbox contains the frozen plan alongside the repository, and its prompt gives the request, the obligation checklist, and the judging rule. For each obligation $o \in \mathcal{O}$, it returns one judgment $V(o, P)$ with a brief rationale, using three labels analogous to those of fact verification (Thorne et al., 2018): S if P explicitly supports o or an equivalent valid route, C if P commits to a concrete mechanism incompatible with o, and A otherwise, including missing or overly implicit coverage. A concrete conflict takes precedence over generic preservation language. Appendix J gives its full specification.

Metrics. With $N = |\mathcal{O}|$, and n_{S} and n_{C} the numbers of supported and conflicting obligations, $\mathrm{Cov}(P)$ combines Correct Coverage, the supported fraction of atomic units (Min et al., 2023; Lee et al., 2025), and Conflict Share into Plan Coverage:

$$\mathrm{Cor}(P) = \frac{n_{\mathrm{S}}}{N}, \qquad \mathrm{Err}(P) = \frac{n_{\mathrm{C}}}{N}, \qquad \mathrm{Cov}(P) = 100 \times \mathrm{Cor}(P)\big(1 - \mathrm{Err}(P)\big). \tag{3}$$

Plan Coverage rewards explicit coverage of valid obligations and discounts otherwise broad plans that contain incompatible commitments, so a long plan cannot score well through surface detail.

5 Meta-Evaluation

We validate the scores of **`SWE-Plan`** in two ways. Section 5.1 tests whether the User Simulator and the Plan Judge agree with human experts, and Section 5.2 tests whether Plan Coverage predicts downstream execution success and thus reflects plan quality that matters for implementation. Appendix L details both analyses.

5.1 Evaluator Reliability

Setup. Both evaluators are validated against human labels on 46 **`SWE-Plan`** tasks disjoint from their memories, using 157 questions from these tasks and 353 HiL-Bench questions with official labels for the User Simulator, and every obligation of 138 plans, three per task, for the Plan Judge. Each is compared with a Base variant, a single model call without repository access or retrieved precedents, using DeepSeek-V4-Flash-0731 (DS-V4-F) and DeepSeek-V4-Pro-Preview (DS-V4-P) (DeepSeek, 2026), MiniMax-M3 (MM-M3) (MiniMaxAI, 2026), and HY3 (Tencent, 2026) as backbones. We report Cohen's κ (Cohen, 1960) as the primary metric, with positive-class F1 for the User Simulator and macro-F1 and the plan-level Spearman ρ of Plan Coverage for the Plan Judge.

Table 1: Agreement with human labels. ↑: relative gain (%) over Base.

	User Simulator						Plan Judge					
	HiL-Bench F1		SWE-Plan F1		SWE-Plan κ		κ		Macro-F1		ρ(Cov.)	
Backbone	Base	Ours$_{\Delta(\%)}$	Base	Ours$_{\Delta(\%)}$	Base	Ours$_{\Delta(\%)}$	Base	Ours$_{\Delta(\%)}$	Base	Ours$_{\Delta(\%)}$	Base	Ours$_{\Delta(\%)}$
DS-V4-F	91.4	98.0$_{\uparrow 7.2}$	83.5	93.0$_{\uparrow 11.4}$	0.74	0.89$_{\uparrow 20.3}$	0.53	0.76$_{\uparrow 43.4}$	0.64	0.80$_{\uparrow 25.0}$	0.75	0.91$_{\uparrow 21.3}$
DS-V4-P	90.1	96.8$_{\uparrow 7.4}$	80.7	90.2$_{\uparrow 11.8}$	0.69	0.85$_{\uparrow 23.2}$	0.50	0.69$_{\uparrow 38.0}$	0.58	0.77$_{\uparrow 32.8}$	0.73	0.86$_{\uparrow 17.8}$
MM-M3	86.5	93.6$_{\uparrow 8.2}$	80.4	90.3$_{\uparrow 12.3}$	0.67	0.84$_{\uparrow 25.4}$	0.46	0.72$_{\uparrow 56.5}$	0.61	0.74$_{\uparrow 21.3}$	0.70	0.88$_{\uparrow 25.7}$
HY3	84.2	91.8$_{\uparrow 9.0}$	80.6	87.5$_{\uparrow 8.6}$	0.71	0.80$_{\uparrow 12.7}$	0.49	0.67$_{\uparrow 36.7}$	0.57	0.75$_{\uparrow 31.6}$	0.74	0.83$_{\uparrow 12.2}$

Results. Table 1 shows that repository access and retrieved precedents improve every metric over Base across all backbones, raising κ by 12.7% to 56.5%, with DS-V4-F achieving the strongest agreement. The ordering of the other three backbones is not the same in every column. With DS-V4-F, the User Simulator reaches $\kappa = 0.89$ on **SWE-Plan** and an F1 of 98.0 on HiL-Bench, and the Plan Judge reaches $\kappa = 0.76$ with a plan-level ρ of 0.91. Consistent gains on an external benchmark and on our own tasks indicate calibrated judgment boundaries rather than memorization of **SWE-Plan** tasks, and the plan-level agreement shows that residual disagreements on individual obligations do not distort plan scores. On plans from DeepSeek-V4.1-Flash (DS-V4.1-F) (DeepSeek, 2026), which shares the backbone's model family, the Plan Judge reaches $\kappa = 0.74$, comparable to 0.76 overall. The User Simulator shows the same pattern, with $\kappa = 0.88$ on DS-V4.1-F pairs and 0.89 on the others. Both evaluators are therefore reliable enough to replace human judgment in **SWE-Plan**, and we adopt DS-V4-F as their backbone.

5.2 Predictive Validity

Setup. We sample three plans for each of 100 tasks so that Plan Coverage varies within each task and spans its full range across tasks, for 300 plans total. Since the plans of a task share its difficulty, comparing them separates plan quality from task difficulty. An executor implements each task given the rewritten request alone, the request with all registered answers, or the request with the answers and each plan. To isolate verification, we additionally execute each plan with its verification content removed and everything else unchanged. We use four executors, GPT-5.6-Sol (GPT-5.6-S) (OpenAI, 2026c) and GPT-5.6-Luna (GPT-5.6-L) (OpenAI, 2026b) in Codex, and Claude-Opus-5 (Opus-5) and Claude-Sonnet-5 (Sonnet-5) (Anthropic, 2026b) in Claude Code. The repository revision, budget, and evaluation tests are fixed across conditions, plan identities are blinded, and a run succeeds when all evaluation tests pass. Execution outcomes never enter any plan score.

Table 2: Predictive validity of Plan Coverage. ↑: gain over the previous column; ↓: drop from +Plan.

	Execution success (%)				Within-task AUC			
Executor	Request	+Answers	+Plan	w/o Verif.	Length	Holistic	Cor.	Cov.
GPT-5.6-S	34.0	72.0$_{\uparrow 38.0}$	83.3$_{\uparrow 11.3}$	75.3$_{\downarrow 8.0}$	0.55	0.68	0.76	0.81
GPT-5.6-L	26.0	61.0$_{\uparrow 35.0}$	72.7$_{\uparrow 11.7}$	63.0$_{\downarrow 9.7}$	0.53	0.66	0.74	0.79
Opus-5	38.0	77.0$_{\uparrow 39.0}$	85.0$_{\uparrow 8.0}$	78.3$_{\downarrow 6.7}$	0.58	0.72	0.79	0.84
Sonnet-5	29.0	66.0$_{\uparrow 37.0}$	76.0$_{\uparrow 10.0}$	67.3$_{\downarrow 8.7}$	0.54	0.70	0.80	0.86

Results. Table 2 shows the same direction for all four executors. The registered answers raise execution success over the request alone by 35.0 to 39.0 points, confirming that the withheld decisions are required for a correct implementation, and the plan adds another 8.0 to 11.7 points on top of the same answers, a gain attributable to planning alone, with 95% task-bootstrap intervals excluding zero for both gains for every executor. Success stays below saturation even with the full plan, as the tasks are screened to require coordinating several requirements and each condition is a single execution under a fixed budget. Removing verification lowers success by 6.7 to 9.7 points, although the ablated plan stays above the answers alone. Within each task, where difficulty is held fixed, Plan Coverage ranks successful executions above failed ones with an AUC of 0.79 to 0.86 (permutation $p < 0.001$), exceeding plan length and a holistic 1–10 rating by the same backbone without the checklist, and matching or exceeding Correct Coverage. The checklist thus credits plan content rather than surface detail. Success also rises across Plan Coverage bins for every executor, from 43.3% to 56.7% below 60 to 86.7% to 100.0% for perfect plans, with most of the gain from leaving the lowest bin, so higher Plan Coverage translates into better implementations regardless of the executor.

6 EXPERIMENTS

Setup. We evaluate Opus-5, GPT-5.6-S, Gemini-3.8-Flash (Gem-3.8) (DeepMind, 2026), Qwen3.8-Max (Qwen-3.8) (Qwen, 2026), DS-V4.1-F, GLM-5.3 (Z.ai, 2026), Kimi-K3 (Moonshot, 2026), and HY3 in the native Plan Mode of OpenCode, Codex, and Claude Code on all 392 tasks, yielding 24 agent configurations that retain each product's native prompt, tool policy, and turn protocol. The interaction ends when the agent submits its final plan, and implementation is never requested. Both evaluators use DS-V4-F as the backbone, and harness comparisons are paired within model and task, with uncertainty estimated by 10,000 paired task bootstrap resamples. Appendices K, M, and N report the hyperparameters, complete results, and statistical significance of all paired harness comparisons. Our key ***Obs***ervations are presented below.

Table 3: Clarification results (%) of 8 models in the native Plan Mode of three harnesses.

	OpenCode				Codex				Claude Code			
Model	Rate	Prec.	Rec.	F1	Rate	Prec.	Rec.	F1	Rate	Prec.	Rec.	F1
Opus-5	**99.7**	72.6	**94.3**	82.1	98.7	73.8	93.1	82.3	**100.0**	71.9	95.8	82.1
GPT-5.6-S	77.6	86.1	77.6	81.6	75.0	91.2	66.7	77.0	90.1	88.0	81.5	84.6
Gem-3.8	89.0	**92.3**	80.3	85.9	92.6	**97.0**	83.6	**89.8**	98.7	**91.2**	89.5	**90.3**
Qwen-3.8	96.2	79.9	90.1	84.7	**100.0**	69.9	**97.5**	81.4	98.7	84.9	92.7	88.6
DS-V4.1-F	95.9	85.1	88.1	**86.5**	94.9	75.1	89.5	81.7	**100.0**	75.3	95.2	84.1
GLM-5.3	95.9	75.0	89.0	81.4	84.9	87.2	77.8	82.2	**100.0**	73.3	**96.6**	83.4
Kimi-K3	88.5	87.4	80.8	84.0	86.0	89.6	79.5	84.2	96.2	85.5	86.2	85.9
HY3	93.9	87.4	82.3	84.8	91.3	88.9	77.8	83.0	97.4	89.0	86.4	87.7
Overall	**92.1**	**83.2**	**85.3**	**83.9**	**90.4**	**84.1**	**83.2**	**82.7**	**97.6**	**82.4**	**90.5**	**85.8**

***Obs.*❶ The leaderboard resolves agent differences hidden by outcome-level evaluation.** Tables 3 and 4 separate the 24 agent configurations, with Plan Coverage ranging from 75.0 to 93.3 and 36.1% of obligations receiving different judgments across configurations, so the benchmark is not saturated. Figure 4a compares the two dimensions with external coding benchmarks, whose public scores and sources are listed in Appendix O. Because public scores come from heterogeneous scaffolds, we compare only the direction of model rankings. Planning follows external rankings at $\rho = 0.77$, consistent with a coding capability component that outcome-level scores reflect but cannot isolate, whereas clarification follows the opposite direction at $\rho = -0.60$.

Table 4: Planning results (%) of the final plans, judged against the obligation checklists.

	OpenCode			Codex			Claude Code		
Model	Cor.	Err.↓	Cov.	Cor.	Err.↓	Cov.	Cor.	Err.↓	Cov.
Opus-5	**92.0**	3.9	89.2	88.4	4.4	84.7	87.0	4.8	83.2
GPT-5.6-S	91.0	**1.7**	**89.8**	89.2	**2.4**	87.3	**94.6**	**1.6**	**93.3**
Gem-3.8	83.4	5.0	80.0	84.3	2.5	82.5	87.9	3.8	85.2
Qwen-3.8	90.1	4.7	86.7	**90.7**	4.0	**87.7**	84.8	4.0	82.0
DS-V4.1-F	87.6	5.2	83.9	89.3	3.8	86.5	88.7	3.9	86.0
GLM-5.3	88.1	5.7	84.1	84.0	5.2	81.0	85.2	6.6	80.8
Kimi-K3	86.2	6.2	82.1	85.7	4.3	82.7	81.5	7.2	76.7
HY3	81.4	7.3	76.8	80.5	7.5	76.0	79.8	8.2	75.0
Overall	**87.5**	**5.0**	**84.1**	**86.5**	**4.3**	**83.6**	**86.2**	**5.0**	**82.8**

***Obs.*❷ Clarification and planning are coupled but not interchangeable.** Figure 4b shows that, within a fixed model and task, changes in Blocker Recall across harnesses are positively correlated with changes in Plan Coverage at $r = 0.62$, so recovering a withheld requirement is associated with a better plan. Across models, however, the two rankings do not align, as the within-harness rank correlation between Ask-F1 and Plan Coverage ranges from -0.71 to -0.14. Tables 3 and 4 show that the strongest clarifier is not the strongest planner. *Clarification can unlock planning but cannot substitute for it, so the two dimensions should be analyzed separately.*

***Obs.*❸ Failures in both dimensions arise from what is not explicitly stated.** Agents handle explicit information well. In a 60-case pilot, final plans incorporate 98.0% to 99.2% of answered blockers exactly; behavior obligations receive 91.2% support, and 93.4% of sessions ask at least one question. Failures concentrate instead on what no input states, and Figure 5 shows this split in both dimensions. Clarification achieves 86.3% mean Blocker Recall despite this readiness to ask. In planning, verification, which no answer requests, is the least supported role for every model in

Table 5, and stronger models narrow the verification gap from 43.2 to 19.6 points. The gap still matters downstream, since removing verification from otherwise unchanged plans lowers execution success by 6.7 to 9.7 points, so tests added during implementation do not substitute for verification designed in the plan. *The weakness of agents is a shared deficit in identifying implicit requirements and obligations, visible when both dimensions are measured within the same interaction.*

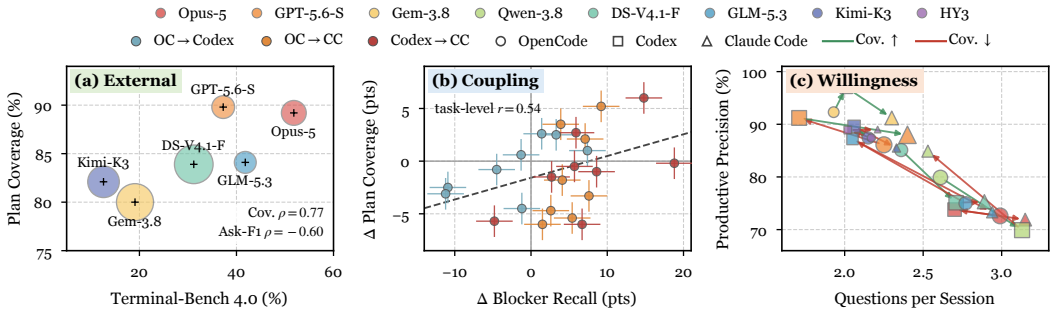


Figure 4: (a) Plan Coverage against Terminal-Bench 4.0 (Merrill et al., 2026), sized by Ask-F1. (b) Paired changes with 95% intervals. (c) Willingness versus precision, sized by Plan Coverage.

***Obs.*❹ Clarification and planning must be evaluated and optimized jointly across models and harnesses.** Question precision and planning vary mainly by model, while harness changes affect willingness more than precision. Plan Coverage and Productive Precision are anchored in the model, with ranges across model means of 14.2 and 20.7 points against 1.3 and 1.7 across harness means, and model rankings stay stable across harnesses at $\rho = 0.63$ and 0.83. Willingness varies with both, and the harness range in Ask Rate, 7.2 points, is four times that of Productive Precision. At the harness level, Claude Code raises the Ask Rate by 5.5 points and Questions per Session by 0.19 over OpenCode, while Productive Precision changes by only 0.8 points, as shown in Figure 4c. At the model level, GLM-5.3 asks 2.77 questions per session at 75.0% precision, whereas GPT-5.6-S asks 2.25 at 86.1% and reaches higher Plan Coverage, 89.8 versus 84.1. The two roles are nonetheless interdependent, since a harness can raise how much agents ask, but given the coupling in Obs.❷, the extra questions pay off only when the model asks accurately and plans well. Harness effects on plans therefore differ in direction across models and track each model's Productive Precision at $r = 0.47$. *Pre-implementation capability is thus a joint property of the model and the harness across both dimensions, which no single component or metric can capture, so advancing coding agents, planners, and orchestrators requires evaluating and optimizing this phase as a whole.*

Table 5: Obligation support rates by role (%). Gap is the mean of the other four roles minus verification.

Model	Behav.	Cond.	Own.	Pres.	Verif.	Gap↓
Opus-5	94.4	**93.4**	**95.1**	**97.8**	73.5	21.7
GPT-5.6-S	**96.3**	92.3	93.2	94.3	**74.4**	**19.6**
Gem-3.8	87.8	87.1	86.5	92.1	52.1	36.3
Qwen-3.8	92.9	91.4	91.5	97.3	69.2	24.1
DS-V4.1-F	91.5	90.9	91.8	95.7	58.5	34.0
GLM-5.3	91.2	91.3	89.9	96.0	62.5	29.6
Kimi-K3	88.9	89.2	89.3	95.5	58.9	31.8
HY3	86.3	84.1	84.6	93.7	43.9	43.2
Overall	**91.2**	**90.0**	**90.2**	**95.3**	**61.6**	**30.0**

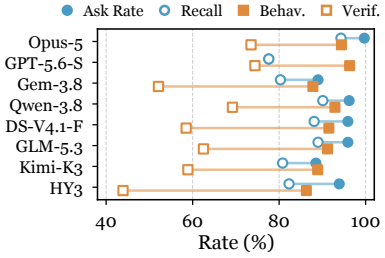


Figure 5: Explicit (filled) versus implicit (hollow) quantities by model (%).

7 Conclusion

We introduced **SWE-Plan**, the first benchmark that isolates the complete pre-implementation phase of coding agents from execution and evaluates it directly. It measures clarification and planning separately through execution-decoupled evaluation and reveals a shared weakness of current agents in analyzing implicit requirements. Grounded in software engineering research on pre-implementation, **SWE-Plan** turns this phase from an implicit part of end-to-end outcomes into a capability that can be measured and attributed independently. We believe this work paves the way for optimizing coding agents, dedicated planner models, and orchestrators.

AI USE STATEMENT

LLMs are the object of study in this work. The eight evaluated models are LLMs operating within coding-agent harnesses, and the User Simulator and Plan Judge are LLM-based agents. Both roles are part of the experimental method, and we validate both judges against human experts. During benchmark construction, LLM-based agents assisted in two steps. SWE-Factory (Guo et al., 2026) built the candidate environments and base tasks from real repositories, and three coding agent configurations enumerated candidate implementation routes from which the experts consolidated the obligation checklists. The source issues, developer patches, and tests come from real software development rather than synthetic generation, and every retained task, blocker, checklist, and memory case was independently reviewed by two human experts. In preparing the manuscript, LLMs were used to improve English grammar, sentence structure, clarity, and flow, and to help write the plotting scripts for the figures. LLMs were also used to retrieve and discover potentially relevant papers, official documentation, and other sources. The authors subsequently examined the original sources and verified their relevance, bibliographic information, and the specific claims supported by each citation. LLMs did not determine the core scientific questions, experimental design, gold labels, or conclusions of this work. We reviewed all AI-assisted material and take responsibility for the final content.

ETHICS STATEMENT

SWE-Plan is constructed from 154 public open-source repositories and their public issue and pull-request histories, and its use of these materials follows each repository's license. No private or proprietary data is involved. Task requests are rewritten by the experts to withhold the registered decisions, and the released tasks contain no personal information beyond what already appears in the public repositories. The benchmark evaluates the questions and plans produced by model–harness configurations and does not assess the developers associated with the source issues or pull requests. All task screening, blocker review, memory annotation, checklist consolidation, and validation labeling were conducted by members of the author team, and no crowdworkers or external annotators were employed. The authors declare no conflicts of interest.

REPRODUCIBILITY STATEMENT

The supplementary materials contain the full benchmark, evaluation harness, and analysis scripts. The paper and its appendices document the task formulation; the construction pipeline and its screening, blocker, and acceptance criteria; the memory annotation protocol; the User Simulator and Plan Judge, including their prompts; the checklist design; the harness configurations and hyperparameters; the evaluator validation and execution protocol; and the complete results with bootstrap intervals. All 392 tasks are released with their blocker registries, obligation checklists, and container specifications, together with the judge memories. Our work is accessible at https://anonymous.4open.science/r/swe-plan-F8D8/.

REFERENCES

Jean-Raymond Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

Anthropic. How we built our multi-agent research system. https://www.anthropic.com/engineering/multi-agent-research-system, 2025. Anthropic Engineering blog.

Anthropic. Claude Code CLI reference. Official documentation, 2026a. URL https://code.claude.com/docs/en/cli-usage. Accessed 2026-09-23.

Anthropic. Anthropic's transparency hub. https://www.anthropic.com/transparency, 2026b. Official model report, accessed 2026-09-23.

Ramakrishna Bairi, Atharv Sonwane, Aditya Kanade, Vageesh D C, Arun Shankar Iyer, Suresh Parthasarathy, Sriram K. Rajamani, B. Ashok, and Shashank Shet. CodePlan: Repository-level coding using LLMs and planning. *Proceedings of the ACM on Software Engineering*, 1(FSE): 675–698, 2024. doi: 10.1145/3643757. URL https://doi.org/10.1145/3643757.

Victor R. Basili, Scott Green, Oliver Laitenberger, Filippo Lanubile, Forrest Shull, Sivert Sorumgard, and Marvin V. Zelkowitz. The empirical investigation of perspective-based reading. *Empirical Software Engineering*, 1(2):133–164, 1996. doi: 10.1007/BF00368702.

Barry Boehm and Victor R. Basili. Software defect reduction top 10 list. *Computer*, 34(1):135–137, 2001. doi: 10.1109/2.962984.

Barry W. Boehm. *Software Engineering Economics*. Prentice-Hall, 1981.

Barry W. Boehm. A spiral model of software development and enhancement. *Computer*, 21(5): 61–72, 1988. doi: 10.1109/2.59.

Alexander Borgida, John Mylopoulos, and Raymond Reiter. On the frame problem in procedure specifications. *IEEE Transactions on Software Engineering*, 21(10):785–798, 1995.

Frederick P. Brooks, Jr. No silver bullet: Essence and accidents of software engineering. *Computer*, 20(4):10–19, 1987.

Neil Chowdhury, James Aung, Chan Jun Shern, Oliver Jaffe, Dane Sherburn, Giulio Starace, Evan Mays, Rachel Dias, Marwan Aljubeh, Mia Glaese, Carlos E. Jimenez, John Yang, Leyton Ho, Tejal Patwardhan, Kevin Liu, and Aleksander Madry. Introducing SWE-bench Verified. OpenAI, 2024. URL https://openai.com/index/introducing-swe-bench-verified/.

Michael G. Christel and Kyo C. Kang. Issues in requirements elicitation. Technical Report CMU/SEI-92-TR-012, Software Engineering Institute, Carnegie Mellon University, 1992.

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

Cursor. Introducing plan mode. Cursor Product Blog, 2025. URL https://cursor.com/blog/plan-mode. Accessed 2026-09-23.

DeepMind. Gemini 3.8 flash model card. https://deepmind.google/models/model-cards/gemini-3-8-flash/, 2026. Official model card, accessed 2026-09-23.

DeepSeek. Deepseek-v4 model card. https://fe-static.deepseek.com/chat/transparency/deepseek-V4-model-card-EN.pdf, 2026. Official model card, accessed 2026-09-23.

DeepSeek-AI. DeepSeek-V4.1-Flash model card. https://huggingface.co/deepseek-ai/DeepSeek-V4.1-Flash, 2026. Accessed 2026-09-26.

Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, Karmini Sampath, Maya Krishnan, Srivatsa Kundurthy, Sean Hendryx, Zifan Wang, Vijay Bharadwaj, Jeff Holm, Raja Aluri, Chen Bo Calvin Zhang, Noah Jacobson, Bing Liu, and Brad Kenstler. SWE-Bench Pro: Can AI agents solve long-horizon software engineering tasks? *arXiv preprint arXiv:2509.16941*, 2025. URL https://arxiv.org/abs/2509.16941.

Lutfi Eren Erdogan, Nicholas Lee, Sehoon Kim, Suhong Moon, Hiroki Furuta, Gopala Anumanchipalli, Kurt Keutzer, and Amir Gholami. Plan-and-act: Improving planning of agents for long-horizon tasks. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 15419–15462, 2025.

Michael E. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3):182–211, 1976. doi: 10.1147/sj.153.0182.

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

Adam Fourney, Gagan Bansal, Hussein Mozannar, Cheng Tan, Eduardo Salinas, Erkang Zhu, Friederike Niedtner, Grace Proebsting, Griffin Bassman, Jack Gerrits, Jacob Alber, Peter Chang, Ricky Loynd, Robert West, Victor Dibia, Ahmed Awadallah, Ece Kamar, Rafah Hosn, and Saleema Amershi. Magentic-One: A generalist multi-agent system for solving complex tasks. *arXiv preprint arXiv:2411.04468*, 2024.

GitHub. Asking GitHub Copilot questions in your IDE. GitHub Docs, 2026. URL https://docs.github.com/en/copilot/how-tos/chat-with-copilot/chat-in-ide. Accessed 2026-09-23.

Lianghong Guo, Yanlin Wang, Caihua Li, Wei Tao, Pengyu Yang, Jiachi Chen, Haoyu Song, Duyu Tang, and Zibin Zheng. SWE data construction, automatically! *Proceedings of the ACM on Software Engineering*, 3(FSE):525–546, 2026. doi: 10.1145/3797119. URL https://doi.org/10.1145/3797119.

Grant Hamblin, Kevin Song, Zhanda Zhu, Anand Jayarajan, Sihang Liu, Nandita Vijaykumar, and Gennady Pekhimenko. SpecBench: Evaluating specification-level reasoning for software engineering LLM agents. *arXiv preprint arXiv:2605.30314*, 2026. URL https://arxiv.org/abs/2605.30314.

ISO. ISO/IEC/IEEE 12207:2017 systems and software engineering – software life cycle processes. Technical report, International Organization for Standardization, 2017.

ISO. ISO/IEC/IEEE 29148:2018 systems and software engineering – life cycle processes – requirements engineering. Technical report, International Organization for Standardization, 2018.

Zimo Ji, Zekai Zhang, Congying Xu, Yujia Tian, Zongjie Li, Yudong Gao, Shuai Wang, and Shing-Chi Cheung. Coding agents are guessing: Measuring action-boundary violations in underspecified DevOps instructions. *arXiv preprint arXiv:2607.02294*, 2026. URL https://arxiv.org/abs/2607.02294.

Bowen Jiang, Haowei Cheng, Yuhong Fu, Anne Koziolek, Jialong Li, and Weixing Zhang. Requirements after the first edit: Mining late requirement emergence and rework in real-world coding-agent sessions. *arXiv preprint arXiv:2609.03028*, 2026.

Xue Jiang, Yihong Dong, Lecheng Wang, Zheng Fang, Qiwei Shang, Ge Li, Zhi Jin, and Wenpin Jiao. Self-planning code generation with large language models. *ACM Transactions on Software Engineering and Methodology*, 33(7):182:1–182:30, 2024. doi: 10.1145/3672456.

Zhonghao Jiang, David Lo, and Zhongxin Liu. Agentic software issue resolution with large language models: A survey. *arXiv preprint arXiv:2512.22256*, 2025.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R. Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

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

Yukyung Lee, JoongHoon Kim, Jaehee Kim, Hyowon Cho, Jaewook Kang, Pilsung Kang, and Najoung Kim. CheckEval: A reliable LLM-as-a-judge framework for evaluating text generation using checklists. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 15771–15798, Suzhou, China, 2025. Association for Computational Linguistics. doi: 10.18653/v1/2025.emnlp-main.796. URL https://aclanthology.org/2025.emnlp-main.796/.

Bowen Li, Wenhan Wu, Ziwei Tang, Lin Shi, John Yang, Jinyang Li, Shunyu Yao, Chen Qian, Binyuan Hui, Qicheng Zhang, Zhiyin Yu, He Du, Ping Yang, Dahua Lin, Chao Peng, and Kai Chen. DevBench: A comprehensive benchmark for software development. *arXiv preprint arXiv:2403.08604*, 2024.

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

Shuhan Liu, Zhiyi Zhao, Xing Hu, Kui Liu, Xiaohu Yang, and Xin Xia. RACE-Bench: A reasoning-augmented benchmark for repository-level code agents on feature addition. *arXiv preprint arXiv:2603.26337*, 2026. URL https://arxiv.org/abs/2603.26337.

Hanjun Luo, Shenyu Dai, Chiming Ni, Xinfeng Li, Guibin Zhang, Kun Wang, Tongliang Liu, and Hanan Salam. AgentAuditor human-level safety and security evaluation for LLM agents. In *Advances in Neural Information Processing Systems*, volume 38, 2025. doi: 10.52202/085713-1440.

Mike A. Merrill, Alexander G. Shaw, Nicholas Carlini, et al. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=a7Qa4CcHak.

Sewon Min, Kalpesh Krishna, Xinxi Lyu, Mike Lewis, Wen-tau Yih, Pang Wei Koh, Mohit Iyyer, Luke Zettlemoyer, and Hannaneh Hajishirzi. FActScore: Fine-grained atomic evaluation of factual precision in long form text generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 12076–12100, 2023.

MiniMaxAI. Minimax-m3, 2026. URL https://huggingface.co/MiniMaxAI/MiniMax-M3. Accessed 2026-09-15.

Moonshot. Kimi k3. https://www.kimi.com/en/blog/kimi-k3, 2026. Official technical blog, accessed 2026-09-23.

Fangwen Mu, Lin Shi, Song Wang, Zhuohao Yu, Binquan Zhang, Chenxue Wang, Shichao Liu, and Qing Wang. ClarifyGPT: A framework for enhancing LLM-based code generation via requirements clarification. *Proceedings of the ACM on Software Engineering*, 1(FSE):2332–2354, 2024. doi: 10.1145/3660810. URL https://doi.org/10.1145/3660810.

Bashar Nuseibeh and Steve Easterbrook. Requirements engineering: A roadmap. In *Proceedings of the Conference on the Future of Software Engineering*, pp. 35–46. Association for Computing Machinery, 2000. doi: 10.1145/336512.336523.

Zach Nussbaum, John X. Morris, Brandon Duderstadt, and Andriy Mulyar. Nomic Embed: Training a reproducible long context text embedder. *arXiv preprint arXiv:2402.01613*, 2024. URL https://arxiv.org/abs/2402.01613.

OpenAI. Run long horizon tasks with Codex. OpenAI Developers, 2026a. URL https://developers.openai.com/blog/run-long-horizon-tasks-with-codex. Accessed 2026-09-23.

OpenAI. GPT-5.6 luna model. https://developers.openai.com/api/docs/models/gpt-5.6-luna, 2026b. Official model documentation, accessed 2026-09-26.

OpenAI. GPT-5.6 sol model. https://developers.openai.com/api/docs/models/gpt-5.6-sol, 2026c. Official model documentation, accessed 2026-09-23.

OpenCode. Agents. https://opencode.ai/docs/agents/, n.d. Official documentation, accessed 2026-09-22.

David L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

David L. Parnas and David M. Weiss. Active design reviews principles and practices. In *Proceedings of the 8th International Conference on Software Engineering*, pp. 132–136. IEEE Computer Society Press, 1985.

David Lorge Parnas and Paul C. Clements. A rational design process: How and why to fake it. *IEEE Transactions on Software Engineering*, SE-12(2):251–257, 1986. doi: 10.1109/TSE.1986.6312940.

Huy N. Phan, Tien N. Nguyen, Phong X. Nguyen, and Nghi D. Q. Bui. HyperAgent: Generalist software engineering agents to solve coding tasks at scale. *arXiv preprint arXiv:2409.16299*, 2024.

Qwen. Qwen3.8-max. https://www.alibabacloud.com/help/en/model-studio/text-generation-model, 2026. Official model documentation, accessed 2026-09-23.

William E. Riddle and Bryan Edwards. Flight software requirements and design support system. NASA Contractor Report NASA-CR-163425, University of Colorado at Boulder, 1980. URL https://ntrs.nasa.gov/citations/19800021560.

Winston W. Royce. Managing the development of large software systems. In *Proceedings of IEEE WESCON*, pp. 1–9, 1970.

Hongjin Su, Shizhe Diao, Ximing Lu, Mingjie Liu, Jiacheng Xu, Xin Dong, Yonggan Fu, Peter Belcak, Hanrong Ye, Hongxu Yin, Yi Dong, Evelina Bakhturina, Tao Yu, Yejin Choi, Jan Kautz, and Pavlo Molchanov. ToolOrchestra: Elevating intelligence via efficient model and tool orchestration. *arXiv preprint arXiv:2511.21689*, 2025.

Tencent. Hy3. https://huggingface.co/tencent/Hy3, 2026. Official model card, accessed 2026-09-23.

Terminal-Bench Team. Terminal-bench 4.0 leaderboard. https://www.tbench.ai/leaderboard, 2026. Accessed 2026-09-26.

James Thorne, Andreas Vlachos, Christos Christodoulopoulos, and Arpit Mittal. FEVER: A large-scale dataset for fact extraction and VERification. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 809–819, 2018.

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

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

Kaixin Wang, Tianlin Li, Xiaoyu Zhang, Chong Wang, Weisong Sun, Yang Liu, and Bin Shi. Software development life cycle perspective: A survey of benchmarks for code large language models and agents. *arXiv preprint arXiv:2505.05283*, 2025. doi: 10.48550/arXiv.2505.05283. URL https://arxiv.org/abs/2505.05283.

Hironori Washizaki (ed.). *Guide to the Software Engineering Body of Knowledge (SWEBOK Guide), Version 4.0*. IEEE Computer Society, 2024. URL https://www.computer.org/education/bodies-of-knowledge/software-engineering.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik R. Narasimhan, and Ofir Press. SWE-agent: Agent–computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems*, volume 37, 2024. doi: 10.52202/079017-1601. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/5a7c947568c1b1328ccc5230172e1e7c-Abstract-Conference.html.

John Yang, Carlos E. Jimenez, Alex L. Zhang, Kilian Lieret, Joyce Yang, Xindi Wu, Ori Press, Niklas Muennighoff, Gabriel Synnaeve, Karthik R. Narasimhan, Diyi Yang, Sida I. Wang, and Ofir Press. SWE-bench Multimodal: Do AI systems generalize to visual software domains? In *International Conference on Learning Representations*, 2025. URL https://arxiv.org/abs/2410.03859.

Yanwei Yue, Guibin Zhang, Boyang Liu, Guancheng Wan, Kun Wang, Dawei Cheng, and Yiyan Qi. MasRouter: Learning to route LLMs for multi-agent systems. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 15549–15572, 2025. doi: 10.18653/v1/2025.acl-long.757.

Z.ai. Glm-5.3. https://huggingface.co/zai-org/GLM-5.3, 2026. Official model card, accessed 2026-09-23.

Pamela Zave and Michael Jackson. Four dark corners of requirements engineering. *ACM Transactions on Software Engineering and Methodology*, 6(1):1–30, 1997. doi: 10.1145/237432.237434.

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. SWE-RPG: A unified issue resolution benchmark for requirement clarification, planning, and code generation for coding agents. *arXiv preprint arXiv:2608.09072*, 2026. URL https://arxiv.org/abs/2608.09072.

CONTENTS OF APPENDIX

A Limitations **18**

B Dataset Statistics **18**

C Repository and Task Screening **19**
C.1 Repository Retention . . . 19
C.2 Planning-Necessity Screening . . . 20
C.3 Temporal Isolation . . . 20

D Blocker Criteria **20**
D.1 Admission Criteria . . . 20
D.2 Exclusions . . . 21

E Annotation and Acceptance Procedure **21**
E.1 Blocker Registry . . . 21
E.2 Request and Image Rewriting . . . 21
E.3 Acceptance and Adjudication . . . 21

F Complete Task Examples **22**
F.1 String Destination Parsing . . . 22
F.2 Glob Patterns in Reload Files . . . 22
F.3 Exact Filename Matching . . . 23

G Memory Construction and Annotation **25**
G.1 Shared Annotation Principle . . . 25
G.2 Memory Banks . . . 25
G.3 Retrieval . . . 25

H User Simulator **25**
H.1 Matching Rule . . . 25
H.2 Output Schema . . . 25
H.3 Reply Protocol . . . 26
H.4 Clarification Metrics . . . 26

I Obligation Checklist Design **26**
I.1 Obligation Roles . . . 26
I.2 Writing Rules . . . 26
I.3 Route Enumeration . . . 27
I.4 Enumeration Prompt . . . 27
I.5 Expert Consolidation and Validation . . . 27

J Plan Judge **28**
J.1 Judging Rule . . . 28
J.2 Output Schema . . . 28

K Experimental Hyperparameters **28**
K.1 Harness Configuration . . . 28
K.2 Coding Agent Hyperparameters . . . 28
K.3 Evaluator Hyperparameters . . . 29

L Evaluator Validation Details **29**
L.1 Validation Data . . . 29
L.2 Bias and Robustness . . . 30
L.3 Execution Protocol . . . 30

M Complete Results **31**
M.1 Per-Configuration Results . . . 31

M.2 Results by Change Category 32
M.3 Quantities behind the Observations 32

N Statistical Significance **33**
N.1 Paired Harness Comparisons 33
N.2 Model Differences 34

O External Coding Benchmarks **34**

P Qualitative Cases **35**
P.1 Case 1: Complete Clarification with Undistinguishing Verification 35
P.2 Case 2: Higher Willingness without Higher Precision 37

A LIMITATIONS

We note several limitations of our work:

❶ **Phase and Interaction Scope.** **`SWE-Plan`** focuses on the pre-implementation phase that delivers a single standardized plan, with the User Simulator returning consistent registered answers to keep evaluation comparable and reproducible. It therefore does not cover requirements that evolve during interaction or implementation, such as users changing their preferences or introducing new constraints. Nor does it evaluate agent behavior separately at different stages of a multi-turn interaction, such as replanning or asking again after partial implementation.

❷ **Task Distribution.** Tasks are built from real repositories with implementations of moderate size, so results may differ for larger tasks or naturally occurring clarification sessions. Python accounts for 364 of the 392 tasks, a concentration shared with SWE-bench and SWE-bench Verified, which consist entirely of Python repositories (Jimenez et al., 2024; Chowdhury et al., 2024). Although this reflects where executable repository tasks are most available, results may still differ for other programming languages.

❸ **Harness Configuration.** We evaluate each harness in its native Plan Mode with default settings, while every model uses its officially recommended inference parameters. This setup reflects out-of-the-box product behavior and keeps comparisons consistent. Customizations common in practice, such as custom system prompts, skills, plugins, or additional MCP tools, are not covered. Accordingly, differences between harnesses are observed at the level of complete configurations and cannot be attributed to any single component.

B DATASET STATISTICS

This section reports the complete statistics of **`SWE-Plan`**. Table 6 gives the composition by programming language and change category, Figure 6 shows the distributions of blockers, planning obligations, and repositories, and Table 7 summarizes the scale of requests and repository snapshots.

Table 6: Composition of **`SWE-Plan`** by programming language and change category. A repository can contribute tasks to several categories, so the repository counts in the lower panel do not sum to the total.

	Tasks	Repositories	Blockers	Obligations
By programming language				
Python	364	134	856	4,571
Rust	11	7	28	145
JavaScript	6	4	16	79
Go	4	4	11	51
TypeScript	4	2	10	53
C++	1	1	2	11
Elixir	1	1	2	13
Scala	1	1	4	14
By change category				
Feature changes	249	119	604	3,140
Bug fixes	50	35	112	630
Compatibility changes	39	32	90	484
Data handling	25	20	56	311
State repairs	17	12	41	221
Performance & reliability	12	11	26	151
Total	**392**	**154**	**929**	**4,937**

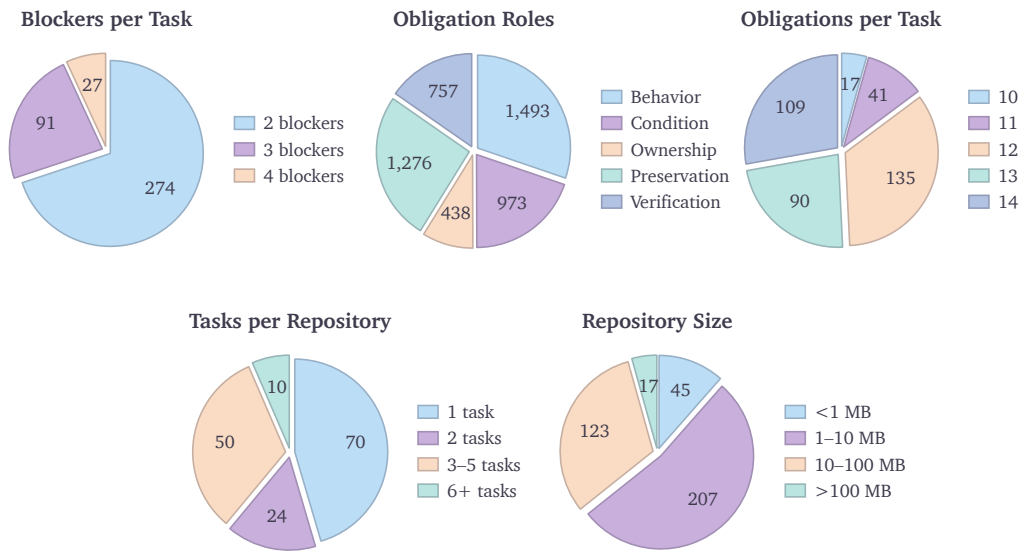


Figure 6: Distributions of blockers, planning obligations, and repositories in **SWE-Plan**. Blocker and obligation counts are aggregated over all tasks, repository contributions count repositories, and snapshot sizes count tasks.

Table 7: Scale of requests and repository snapshots over the 392 tasks. Snapshots refer to the solver-visible repository in the task container.

Statistic	Min	Median	Mean	Max
Request length (words)	15	76	78.6	110
Snapshot files	28	572	1,393	25,732
Snapshot size (MB)	0.1	6.6	19.3	399.7

C Repository and Task Screening

This section details the screening of repositories and base tasks. Each base task pairs an upstream issue with the merged developer patch and the tests of the same repository, aligned to the pinned revision.

C.1 Repository Retention

A candidate repository is retained only if it satisfies all of the following criteria.

- **Active maintenance.** The project shows ongoing maintenance at the pinned revision, so that its code, tests, and issue history reflect current development practice.
- **Continuous integration and reproducible builds.** The project runs active continuous integration, and its dedicated container image builds and runs the test suite at the pinned revision.
- **Human-authored development.** Projects whose history is dominated by AI-generated contributions are excluded, so that the settled decisions reflect the choices of human developers.
- **No agent-specific guidance.** Instruction files intended for coding agents, such as `AGENTS.md` or `CLAUDE.md`, are removed from the solver-visible repository, so that every harness receives the same task context.

C.2 Planning-Necessity Screening

Two experts retain a base task only if it satisfies all of the following criteria. A task that fails any criterion is rejected.

- **Repository exploration.** Resolving the task requires locating and reading code beyond what the issue names, such as call sites, configuration, or existing tests. Issues that point to the exact location of the change are rejected.
- **Coordinated requirements.** The implementation must satisfy more than one behavioral or compatibility requirement at once, such as new behavior together with existing behavior that must be preserved.
- **Open implementation decisions.** The implementation involves at least one decision with more than one plausible resolution. These decisions are the candidates for blocker annotation.
- **Non-trivial scope.** The task cannot be reduced to a local lookup, a single-line completion, or a direct restatement of the issue text.

C.3 Temporal Isolation

Every repository is pinned at a commit later than June 1, 2026. Table 8 lists the published knowledge cutoffs of the evaluated models. Month-level disclosures are converted to the end of the reported month, and every published cutoff precedes this gate. Five models do not disclose a cutoff. For these models, isolation rests on the rewritten requests, blocker registries, and obligation checklists, which were created for **`SWE-Plan`** and have not been released.

Table 8: Published knowledge cutoffs of the evaluated models. ND denotes a cutoff that was not publicly disclosed when the experiment metadata were frozen.

Model	Published knowledge cutoff	Before June 1, 2026
GPT-5.6-Sol (OpenAI, 2026c)	February 16, 2026	✓
Gemini-3.8-Flash (DeepMind, 2026)	March 2026	✓
Claude-Opus-5 (Anthropic, 2026b)	May 2026	✓
DeepSeek-V4.1-Flash (DeepSeek, 2026)	ND	–
GLM-5.3 (Z.ai, 2026)	ND	–
HY3 (Tencent, 2026)	ND	–
Kimi-K3 (Moonshot, 2026)	ND	–
Qwen3.8-Max (Qwen, 2026)	ND	–

D Blocker Criteria

We adopt the notion of blockers from HiL-Bench (Trinh et al., 2026) and adapt its blocker quality criteria to decisions that the developer patch and tests have already settled. A decision is retained as a blocker only if it satisfies all admission criteria and meets no exclusion criterion. Any single violation causes rejection.

D.1 Admission Criteria

- **Repository unresolvability.** The value of the decision cannot be inferred from the request, the source code, the documentation, or the visible tests. The registered answer exists only in the blocker registry.
- **Plausible alternatives.** At least two answers remain plausible to an experienced developer who has read the repository, and the repository offers no basis for choosing between them.
- **Plan criticality.** The plausible answers lead to materially different correct plans, for example in which components change, which behavior is preserved, or which tests are required. Litmus test: two developers who choose different answers would write plans that differ beyond wording.
- **Objectivity.** The developer patch and tests settle a single registered answer, stated precisely enough to check whether a plan follows it.

- **Independence.** Resolving one blocker does not reveal the answer to another blocker of the same task, so each blocker requires its own question.
- **Grounding.** The decision arises from the task itself, and a developer who reads the request and the repository would recognize it as worth asking about.

D.2 Exclusions

- **Cosmetic choices.** Message wording, naming, and formatting change the wording of a plan but not its direction.
- **Conventional defaults.** Choices that an experienced developer would make without asking, such as following the repository's existing error type or configuration format.
- **Repository-settled decisions.** Decisions fixed by existing conventions, documented behavior, or visible tests.
- **Dependent decisions.** Decisions whose answer follows from the answer to another blocker of the same task.

E Annotation and Acceptance Procedure

E.1 Blocker Registry

Each blocker is stored as one registry entry with the fields in Table 9. The User Simulator returns only the registered answer, and the remaining fields support annotation, acceptance, and evaluation.

Table 9: Fields of a blocker registry entry.

Field	Content
Blocker description	The open decision, written without revealing its answer.
Repository insufficiency	Why the request and the repository cannot settle the decision.
Registered answer	The answer settled by the developer patch and tests.
Resolved requirement	The requirement that the registered answer imposes on the implementation.
Effect on the plan	How the plausible answers lead to different plans.
Evidence	Lines of the developer patch and the tests that settle the answer.

E.2 Request and Image Rewriting

- **Answer removal.** Each registered answer is removed from the request and, if necessary, from the container image.
- **Retained context.** The symptoms and scope of the task are kept, so each blocker can still be noticed during exploration. Its answer, however, cannot be inferred.
- **Natural phrasing.** The rewritten request reads as an ordinary maintainer request and remains understandable on its own. Explicit placeholders such as `[REDACTED]` or `TODO` are not used.

E.3 Acceptance and Adjudication

Executability. Each task ships with a container recipe and an evaluation script that applies the hidden tests. A task passes when the edited container builds, and the evaluation tests fail on it and pass after the developer patch is applied. All 392 tasks pass this check.

Planning Necessity. The two experts follow three stages.

- **Independent review.** Each expert checks every task independently, without seeing the other's decisions, against the screening criteria in Appendix C.2, the blocker criteria in Appendix D, and the rewriting rules in Appendix E.2.

- **Structured record.** For each task, the expert records a pass or fail decision for every check, the blockers involved in any failure, and a concise rationale.
- **Adjudication.** The experts compare their records and revisit the request, the repository, and the registry whenever they disagree, and discuss each disagreement until they reach consensus. Tasks that fail any check are revised and reviewed again.

F Complete Task Examples

This section presents three complete tasks with two, four, and four blockers from two programming languages. Registry fields are abridged, while code identifiers are preserved.

F.1 String Destination Parsing

This task withholds two decisions about how a library parses string inputs and reports parse failures.

Task Example 1: adamchainz/time-machine

Task Classification

- **Repository.** `adamchainz/time-machine`, Python.
- **Category.** Feature change, with two blockers.

Rewritten Request

time-machine is establishing the destination parsing contract for `travel()` when the destination arrives as a string. Strings reaching it range from plain timestamps through extended ISO forms to loose prose, and string support is currently bound to an optional third-party dependency. The existing gated branch and the unsupported-destination type error do not determine which parser has first claim on a string, nor which failure a caller sees when no parser accepts it. Before planning, clarify the string destination parser precedence and the unparsable string failure contract. Preserve datetime, date, and numeric destination handling, timezone name extraction, and nested travel behavior.

Blocker 1: String Parser Precedence

- **Open decision.** Whether the standard library's ISO parser or the third-party parser has first claim on a string destination, and whether the third-party parser remains a precondition for string support.
- **Repository insufficiency.** The visible module expresses only the dependency-gated policy, and no visible test drives the parser with the dependency absent.
- **Registered answer.** The ISO parser `dt.datetime.fromisoformat` has first claim, and the third-party parser is consulted only when ISO parsing fails and the dependency is installed.
- **Effect on the plan.** An optional fallback requires the plan to decide the coverage of ISO parsing across supported Python versions, whereas a mandatory front parser confines the plan to error handling.

Blocker 2: Unparsable String Failure

- **Open decision.** Which error reaches the caller when every available parser rejects a string, and whether earlier failures remain attached as context.
- **Repository insufficiency.** The visible module never tries two parsers, so it states no policy on which parse failure is authoritative.
- **Registered answer.** The last parser to refuse owns the failure. With the dependency installed, its `ValueError` is re-raised with the ISO failure suppressed, and without it, the ISO failure surfaces unchanged.
- **Effect on the plan.** The error message then depends on the installed dependencies, so the plan must document it per installation and parameterize its tests over the dependency.

Evidence

The developer patch adds a `fromisoformat` branch for strings and re-raises the third-party error with `from None`. Hidden tests such as `test_destination_string_no_dateutil` and `test_destination_string_unparsable_dateutil` check both answers.

F.2 Glob Patterns in Reload Files

This task withholds four decisions about glob patterns in the files that trigger server reloads.

Task Example 2: benoitc/gunicorn

Task Classification

- **Repository.** `benoitc/gunicorn`, Python.
- **Category.** Feature change, with four blockers.

Rewritten Request

gunicorn is extending `reload_extra_files` so entries may be glob-style patterns as well as literal paths in reload checks. The present validator treats every entry as a file that must already exist; that behavior is implementation history, not an authoritative pattern-versus-literal contract. Before planning, clarify the expansion timing, the pattern-vs-literal grammar, and the recursive-match depth. Preserve existing validation for literal paths that do not use glob syntax.

Blocker 1: Expansion Timing

- **Open decision.** Whether patterns are expanded once at configuration load or re-expanded on every reload check.
- **Repository insufficiency.** One-shot expansion matches the current static file set, while re-expansion covers files created after startup, and the request does not state whether such files are in scope.
- **Registered answer.** Patterns stay unexpanded in the configuration and are re-expanded on every reloader file collection.
- **Effect on the plan.** The reloader must store patterns separately from literal files, and tests must cover files created after startup.

Blocker 2: Pattern Grammar

- **Open decision.** Which syntax marks an entry as a pattern rather than a literal path that must exist.
- **Repository insufficiency.** A star-only rule, the `*?[` set of Python glob, and a `glob:` prefix are all defensible, and the repository defines no public grammar.
- **Registered answer.** An entry containing `*`, `?`, or `[` is a pattern, and every other entry keeps literal-path validation.
- **Effect on the plan.** Configuration validation and the reloader must share one classifier, or legitimate `?` and `[` patterns are rejected as missing paths.

Blocker 3: Recursive Depth

- **Open decision.** Whether `**` matches nested directories.
- **Repository insufficiency.** Python glob recurses only on request, and recursion covers nested trees at the cost of walking large directories on every check.
- **Registered answer.** Patterns are expanded with `glob.glob(..., recursive=True)`.
- **Effect on the plan.** Validation-time checks must use the same expansion depth as the reloader.

Blocker 4: Unmatched Patterns

- **Open decision.** Whether a pattern that matches no files at configuration time is rejected, accepted silently, or accepted with a warning.
- **Repository insufficiency.** An empty match may be a typo or a watch for files created later, and the literal-path validator cannot choose among the three responses.
- **Registered answer.** The pattern is accepted, and a `RuntimeWarning` reports that it currently matches no files.
- **Effect on the plan.** Validation must warn instead of raising, and tests must assert a warning rather than an exception.

Evidence

One or more hidden tests check each answer:

- `test_extra_patterns_expand_on_every_call` (Blocker 1);
- `test_extra_files_and_patterns_are_separated` (Blocker 2);
- `test_recursive_pattern_matches_nested_files` (Blocker 3);
- `test_reload_extra_files_pattern_without_match_warns` (Blocker 4).

F.3 Exact Filename Matching

This task withholds four decisions about a new command-line option that must compose with existing search modes.

Task Example 3: sharkdp/fd

Task Classification

- **Repository.** `sharkdp/fd`, Rust.
- **Category.** Feature change, with four blockers.

Rewritten Request

Add a new `--exact` search option so users can request a whole-subject match instead of fd's usual substring-style filename search. Keep existing searches unchanged when the option is absent. Before implementation, clarify what pattern grammar "exact" uses, which subject is compared when path matching is enabled, how this mode composes or conflicts with the existing pattern-mode flags, and which case policy applies. Update the user-facing CLI documentation and focused regression coverage once those choices are known.

Blocker 1: Pattern Grammar

- **Open decision.** Whether "exact" means an anchored regular expression, an anchored glob, or a literal whole-string comparison.
- **Repository insufficiency.** fd already supports regex, glob, and literal substring modes, so each reading has a precedent but treats metacharacters differently.
- **Registered answer.** The pattern is a literal string that must match the entire subject, so `.`, `(`, and `)` have no special meaning.
- **Effect on the plan.** The plan needs a dedicated path that escapes user input before enforcing whole-string boundaries.

Blocker 2: Match Subject

- **Open decision.** Whether exactness always applies to the basename or follows the existing `--full-path` mode.
- **Repository insufficiency.** Whole-name and whole-path matching are both reasonable meanings, and the new option must choose how it relates to the existing subject flag.
- **Registered answer.** The complete filename is matched by default, and the complete path is matched under `--full-path`.
- **Effect on the plan.** Subject selection stays in the existing full-path machinery, and the documentation must state the distinction.

Blocker 3: Mode Interactions

- **Open decision.** Whether `--exact` is a modifier that combines with other pattern modes or a distinct mode with explicit conflicts.
- **Repository insufficiency.** Existing pattern flags mix conflicts and overrides, so current conventions permit more than one policy.
- **Registered answer.** `--exact` is a distinct mode rejected with `--glob` or `--fixed-strings`, and it keeps literal semantics when combined with `--regex`.
- **Effect on the plan.** The CLI contract must declare these conflicts, and tests must separate rejected combinations from allowed ones.

Blocker 4: Case Policy

- **Open decision.** Whether exact matching is always case-sensitive, always case-insensitive, or inherits fd's smart-case and explicit case flags.
- **Repository insufficiency.** Exactness describes match extent rather than case equality, so the new mode could plausibly override or inherit the existing policy.
- **Registered answer.** fd's ordinary case policy applies, including smart case and the `--case-sensitive` and `--ignore-case` overrides.
- **Effect on the plan.** The exact literal must pass through the shared regex builder rather than a separate byte comparison, with paired case tests under default and explicit settings.

Evidence

Hidden tests use dots and parentheses as literals, supply a full path under `--full-path`, check the rejected and allowed flag combinations, and test case mismatches under default and `--case-sensitive` settings.

G Memory Construction and Annotation

G.1 Shared Annotation Principle

Annotators apply the matching rule of Appendix H to question–blocker pairs and the judging rule of Appendix J to obligation–plan pairs. The procedure comprises three stages.

- **Independent labeling.** Two experts label every case independently, without seeing one another's decisions or any judge prediction.
- **Boundary rationale.** For every case, the experts record the judgment together with a concise rationale that states which part of the question or plan determines the label.
- **Adjudication.** The experts compare their records and revisit the request, the repository, and the compared output whenever they disagree, and discuss every disagreement until they reach consensus. Only consensus records enter the memories or serve as validation references.

G.2 Memory Banks

Each judge has a frozen memory of 40 manually curated cases, written and revised by hand so that together they cover the boundaries of its rule. The memory of the User Simulator covers successful matches, hard negatives, repeated questions, and questions that match several blockers. The memory of the Plan Judge covers explicit support, equivalent valid routes, missing or overly implicit coverage, and concrete conflicts despite generic preservation language. A case stores the task context, the compared question or plan excerpt, the blocker or obligation, the consensus judgment, and the rationale. Prompts, rules, and memories are frozen before evaluation, and memory cases share no tasks with the validation data.

G.3 Retrieval

Both judges use top-K retrieval by embedding similarity, with $K = 5$. The retrieval key of every memory case is embedded in advance. For the User Simulator, the query combines the new question, the blocker under judgment, and the task context. For the Plan Judge, the query combines the obligation under judgment and its task context. The 5 retrieved cases are supplied to the judge as precedents, and the encoder is reported in Appendix K.

H User Simulator

H.1 Matching Rule

The User Simulator judges every question–blocker pair under the following rule.

- **Match.** A question–blocker pair is matched, coded as a binary 1, when a direct answer to the complete question would determine the decision that the blocker represents, given the request, the evidence gathered so far, and the preceding interaction. All other pairs are coded as 0.
- **Multiple matches.** One question may match several blockers, and paraphrases of a decision are accepted.
- **Non-matches.** Topical mentions, generic requests for detail, repeats of matched decisions, and questions that the repository can answer are not matched.

H.2 Output Schema

User Simulator Output Schema

```
{
  "matched_blocker_ids":  ["<blocker id>"]
}
```

An empty list denotes a question that matches no blocker.

H.3 Reply Protocol

For every newly matched blocker, the User Simulator returns the registered answer from the blocker registry, and the session continues. A question that matches no blocker receives a fixed template reply.

H.4 Clarification Metrics

For a session with questions $q_1, \ldots, q_{n_\mathrm{q}}$ in order and registered blockers $\mathcal{B}$, let $m(q_i, b) \in \{0, 1\}$ denote the match between q_i and $b \in \mathcal{B}$, and define

$$M_i = \{\, b \in \mathcal{B} \mid m(q_i, b) = 1 \,\}, \qquad \Delta_i = M_i \setminus \bigcup_{j<i} M_j, \qquad \mathcal{B}^\star = \bigcup_{i=1}^{n_\mathrm{q}} M_i.$$

Question q_i is productive if $\Delta_i \neq \varnothing$, so that $n_\mathrm{prod} = \sum_{i=1}^{n_\mathrm{q}} \mathbb{I}[\Delta_i \neq \varnothing]$. The session-level metrics are

$$\mathrm{Prec} = \frac{n_\mathrm{prod}}{n_\mathrm{q}} \quad \text{if } n_\mathrm{q} > 0, \qquad \mathrm{Rec} = \frac{|\mathcal{B}^\star|}{|\mathcal{B}|},$$

so that a session without questions has no precision and a recall of 0. For a model–harness configuration evaluated on the set $\mathcal{T}$ of tasks with valid sessions, let $n_{\mathrm{q},t}$ be the number of questions in the session of task t and $\mathcal{T}_\mathrm{ask} = \{t \in \mathcal{T} \mid n_{\mathrm{q},t} > 0\}$. Productive Precision is macro-averaged over $\mathcal{T}_\mathrm{ask}$, Blocker Recall over $\mathcal{T}$, and Ask-F1 is their harmonic mean,

$$\overline{\mathrm{Prec}} = \frac{1}{|\mathcal{T}_\mathrm{ask}|} \sum_{t \in \mathcal{T}_\mathrm{ask}} \mathrm{Prec}_t, \qquad \overline{\mathrm{Rec}} = \frac{1}{|\mathcal{T}|} \sum_{t \in \mathcal{T}} \mathrm{Rec}_t, \qquad \mathrm{AskF1} = \frac{2\,\overline{\mathrm{Prec}}\,\overline{\mathrm{Rec}}}{\overline{\mathrm{Prec}} + \overline{\mathrm{Rec}}},$$

where Ask-F1 follows HiL-Bench (Trinh et al., 2026). Precision thus measures how accurately an agent asks once it asks, while silence lowers recall. The two measures of the willingness to ask are

$$\mathrm{AskRate} = \frac{1}{|\mathcal{T}|} \sum_{t \in \mathcal{T}} \mathbb{I}[n_{\mathrm{q},t} > 0], \qquad \mathrm{QPS} = \frac{1}{|\mathcal{T}|} \sum_{t \in \mathcal{T}} n_{\mathrm{q},t},$$

where QPS denotes Questions per Session.

I Obligation Checklist Design

I.1 Obligation Roles

The five roles cover what $P, K_\mathcal{C} \vdash R$ requires of a plan beyond restating the request.

- **Behavior.** The observable change that a requirement in R demands, corresponding to the action part of a requirement statement (ISO, 2018).
- **Condition.** The inputs, states, or configurations under which a behavior applies. Conditions form separate obligations because a plan can state the right behavior while applying it under the wrong conditions.
- **Ownership.** The existing module or boundary that should carry the change. Since $K_\mathcal{C}$ fixes the current decomposition, a plan can satisfy the request locally while placing the change where it breaks that decomposition.
- **Preservation.** Existing behavior and interfaces that the change must leave intact. Requests rarely state them, so they are made explicit as in the frame conditions of procedure specifications.
- **Verification.** Tests that distinguish a correct implementation from its near misses and thereby make the other obligations verifiable in the sense of ISO/IEC/IEEE 29148.

I.2 Writing Rules

- **Necessity.** Every obligation either preserves a correct requirement or excludes a plausible semantic near miss.

- **Route independence.** An obligation states a property rather than a mechanism. Valid alternative routes are recorded as accepted freedoms of the checklist and receive no bonus.
- **Near miss.** Every obligation is paired with a plausible mechanism that violates it, which marks the boundary between support and conflict.
- **Traceability.** Every obligation cites the lines of the request, the developer patch, or the tests that establish it.
- **Exclusion.** Names, formatting, helper structures specific to the developer patch, and speculative extensions are excluded from scoring.

I.3 Route Enumeration

Implementation routes are enumerated by three coding agent configurations, each run through its subscription plan.

- **Claude Code** with Opus-5.
- **Codex** with GPT-5.6-S.
- **Cursor** with Grok-4.6.

The three configurations receive the same prompt and run independently, so none of them sees the routes of another. For every task, a configuration receives the request, the repository at its pinned revision, the resolved requirements, the developer patch, and the tests, and enumerates at least 10 routes with distinct orientations.

I.4 Enumeration Prompt

Route Enumeration Prompt

Role
You are a senior engineer preparing implementation options for a change request in this repository. Work in read-only mode and do not modify any file.
Inputs
You receive the request, the resolved requirements, the developer patch, and the tests. The developer patch is one valid route. Do not treat it as the only one or as the preferred one.
Task
Enumerate at least 10 implementation routes, each of which satisfies every resolved requirement and passes the tests. The routes must differ in orientation rather than in naming or code layout, for example in where the change is placed, which existing mechanism is extended or replaced, how compatibility is kept, how edge conditions are handled, and how the result is verified.
Subagents
Actively delegate. Assign every orientation to a separate subagent, let every subagent explore the repository on its own, and collect the routes they return. Do not obtain a new route by editing an earlier one.
Route report
For every route, report the approach, the files and components it changes, the behaviors and conditions it implements, the existing behavior it preserves, how it would be verified, and its plausible near misses, meaning implementations of the route that look correct but violate a requirement.
Synthesis
After all routes, list the properties that every valid route must satisfy and the choices that differ across routes. Return one section per route, followed by a section of shared properties and a section of free choices.

I.5 Expert Consolidation and Validation

Two experts consolidate the enumerated routes of every task into its checklist through the following steps.

- **Feasibility check.** Every route is checked against the repository, the developer patch, and the tests. Routes that contradict a resolved requirement, cannot pass the tests, or rely on code that does not exist are rejected.

- **Equivalence merging.** Routes that differ only in naming, code layout, or helper structure are merged into one route.
- **Property extraction.** Correctness properties held by every retained route become candidate obligations, are assigned one of the five roles, and are paired with a near miss drawn from the reported routes or written by the experts. Choices on which retained routes differ become accepted freedoms.
- **Review.** The two experts perform these steps independently and resolve disagreements through the same three-stage procedure as task acceptance in Appendix E.3.

Routes come from three model families, and Grok-4.6 is not among the evaluated models. No construction agent sees any evaluated plan.

J Plan Judge

J.1 Judging Rule

The Plan Judge judges every obligation–plan pair under the following rule.

- **S (supported).** The plan explicitly supports the obligation or an equivalent valid route.
- **C (conflict).** The plan commits to a concrete mechanism incompatible with the obligation.
- **A (absent).** All other cases, including missing or overly implicit coverage.
- **Precedence.** A concrete conflict takes precedence over generic preservation language.

J.2 Output Schema

Plan Judge Output Schema

```
{
  "judgments": [{"obligation_id": "<obligation id>",
                 "judgment": "S|A|C",
                 "rationale": "<brief rationale>"}]
}
```

K Experimental Hyperparameters

K.1 Harness Configuration

Each model is evaluated in the native Plan Mode of OpenCode 1.17.13, Codex CLI 0.149.0-alpha.4.1, and Claude Code 2.1.272, and the native question channel of each harness is routed to the User Simulator. The solver-visible bundle of each task contains only the task container and the rewritten request. The hidden evaluation bundle contains the blocker registry, the registered answers, the developer patch, the evaluation tests, and the runtime scripts.

K.2 Coding Agent Hyperparameters

Every model uses the official inference settings recommended for agentic coding, with high reasoning effort and the largest context window and output limit supported by its interface. Temperature and top-p use the provider defaults unless the provider specifies agentic values. Table 10 reports these settings.

Table 10: Inference settings of the evaluated models. Context and maximum output are in tokens, and – denotes the provider default.

Model	Reasoning	Temp.	Top-p	Context	Max output
Claude-Opus-5	high	–	–	1,000,000	128,000
GPT-5.6-Sol	high	–	–	1,000,000	128,000
Gemini-3.8-Flash	high	–	–	1,048,576	65,536
Qwen3.8-Max	high	–	–	1,000,000	128,000
DeepSeek-V4.1-Flash	high	–	–	1,000,000	128,000
GLM-5.3	high	–	–	1,000,000	128,000
Kimi-K3	high	1.0	–	1,000,000	128,000
HY3	high	0.9	1.0	196,608	128,000

K.3 Evaluator Hyperparameters

Both judges run on a read-only adaptation of mini-SWE-agent (Yang et al., 2024) and share the same settings, listed in Table 11.

Table 11: Evaluator and retrieval hyperparameters, shared by the User Simulator and the Plan Judge.

Setting	Value
Backbone model	DeepSeek-V4-Flash-0731
Temperature	0
Context window	Official maximum
Maximum output	Official maximum
Memory cases per judge	40
Retrieved cases K	5
Retrieval encoder	`nomic-embed-text-v1.5` (Nussbaum et al., 2024)
Embedding dimension	512

L Evaluator Validation Details

L.1 Validation Data

Table 12 summarizes the validation sets. Each **`SWE-Plan`** question is labeled against every registered blocker of its task, giving 372 question–blocker pairs with 138 positives, and the 138 plans contain 1,740 obligation judgments whose role shares match the full benchmark within 0.1 points. Table 13 gives the counts behind the **`SWE-Plan`** F1 and κ of the User Simulator in Table 1.

Table 12: Validation sets. Label distribution gives the shares of questions matching one, several, or no blocker, and the shares of S, A, and C judgments.

Evaluator	Set	Tasks	Units	Label distribution (%)
User Simulator	HiL-Bench	100	353 questions	100.0 / 0.0 / 0.0
User Simulator	**`SWE-Plan`**	46	157 questions, 372 pairs	56.1 / 14.6 / 29.3
Plan Judge	**`SWE-Plan`**	46	138 plans, 1,740 judgments	82.4 / 13.3 / 4.3

Table 13: Question–blocker counts of the User Simulator on the 372 **SWE-Plan** pairs.

Backbone	Variant	TP	FP	FN	TN
DS-V4-F	Base	114	21	24	213
DS-V4-F	Ours	126	7	12	227
DS-V4-P	Base	113	29	25	205
DS-V4-P	Ours	120	8	18	226
MM-M3	Base	123	45	15	189
MM-M3	Ours	131	21	7	213
HY3	Base	102	13	36	221
HY3	Ours	122	19	16	215

L.2 Bias and Robustness

A judge distorts the leaderboard when it shifts the scores of some models more than others, for instance by favoring outputs of its own family, or when rankings change with the choice of judge. The upper part of Table 14 reports the signed difference between scores from DS-V4-F and from human labels, overall, for DS-V4.1-F, which shares the family of the backbone, and for the other models. The lower part repeats the evaluation of the 24 agent configurations with DS-V4-P, MM-M3, and HY3 as judges and reports the Kendall τ between their rankings and those under DS-V4-F.

Table 14: Bias and robustness under DS-V4-F. Signed bias is evaluator minus human, in match-rate percentage points or Plan Coverage points, with 95% intervals on a common residual scale per column.

	User Simulator (pp)		Plan Judge (points)	
	n	**Bias**	n	**Bias**
Signed bias, evaluator minus human				
All outputs	372	-1.34 $[-3.64, 0.95]$	138	$+0.40$ $[-0.37, 1.17]$
DS-V4.1-F	54	-1.85 $[-7.87, 4.17]$	18	$+1.40$ $[-0.73, 3.53]$
Other models	318	-1.26 $[-3.74, 1.22]$	120	$+0.25$ $[-0.57, 1.07]$
Kendall τ with rankings under DS-V4-F				
DS-V4-P		0.93		0.86
MM-M3		0.81		0.88
HY3		0.84		0.77

L.3 Execution Protocol

The three plans of every task lie in three different Plan Coverage bins. Verification is removed by excising every sentence traced to a verification obligation, and the ablated plan is kept only if the judgments of the Plan Judge change on verification obligations alone, which all 300 ablated plans satisfy. Each executor runs in its default implementation mode with provider-default decoding and a budget of 100 implementation turns, never sees the evaluation tests, and adds or modifies tests in 52% of runs. Plan length counts the words of the plan, and the holistic rating asks the evaluator backbone to rate the plan from 1 to 10 given the request, the registered answers, and the repository, without the obligation checklist. Table 15 reports execution success by Plan Coverage bin. In Table 16, the answer gap is paired across the 100 tasks, and the plan gap compares, within each task, the mean outcome of its three plans with its answers-only run.

Table 15: Execution success rates (%) by Plan Coverage bin.

Plan Coverage	Plans	GPT-5.6-S	GPT-5.6-L	Opus-5	Sonnet-5
$[0, 60)$	60	56.7	46.7	53.3	43.3
$[60, 80)$	80	82.5	70.0	85.0	72.5
$[80, 95)$	100	92.0	82.0	95.0	88.0
100	60	96.7	86.7	100.0	93.3

Table 16: Execution gaps (percentage points) with 95% intervals from 10,000 paired task bootstraps.

Executor	+Answers − Request	+Plan − +Answers
GPT-5.6-S	38.0 [28.0, 48.0]	11.3 [5.7, 17.3]
GPT-5.6-L	35.0 [24.0, 46.0]	11.7 [5.0, 18.3]
Opus-5	39.0 [29.0, 49.0]	8.0 [2.3, 14.0]
Sonnet-5	37.0 [27.0, 47.0]	10.0 [3.3, 16.7]

M Complete Results

This section reports the complete results behind Section 6 without further analysis.

M.1 Per-Configuration Results

Table 17 reports every clarification and planning metric of the 24 agent configurations together with the number of valid sessions and scored plans. Every configuration runs on all 392 tasks, and n_{p} falls below 392 only where a plan could not be scored. The 45 unscored plans, 13 in OpenCode, 12 in Codex, and 20 in Claude Code, either failed scoring or contained no scorable plan. Their sessions still count toward the clarification metrics, while the planning metrics exclude them rather than scoring them as 0. Correct Coverage, Conflict Share, and Plan Coverage are each computed per plan by Eq. (3) and macro-averaged over the scored plans, so the reported Plan Coverage differs from the product of the reported Correct Coverage and one minus the reported Conflict Share. Plan Coverage spans 75.0 to 93.3 and Ask-F1 spans 77.0 to 90.3 across the 24 configurations. Of all 4,937 obligations, 1,782 (36.1%) do not receive the same S, A, or C judgment from all 24 configurations.

Table 17: Complete results of the agent configurations (%). n_{s} and n_{p} count valid sessions and scored plans, and QPS is Questions per Session.

		Clarification						Planning			
Model	**Harness**	n_{s}	Rate	QPS	Prec.	Rec.	F1	n_{p}	Cor.	Err.↓	Cov.
Opus-5	OpenCode	392	99.7	2.99	72.6	94.3	82.1	391	92.0	3.9	89.2
	Codex	392	98.7	2.70	73.8	93.1	82.3	391	88.4	4.4	84.7
	Claude Code	392	100.0	3.15	71.9	95.8	82.1	390	87.0	4.8	83.2
GPT-5.6-S	OpenCode	392	77.6	2.25	86.1	77.6	81.6	390	91.0	1.7	89.8
	Codex	392	75.0	1.71	91.2	66.7	77.0	391	89.2	2.4	87.3
	Claude Code	392	90.1	2.40	88.0	81.5	84.6	390	94.6	1.6	93.3
Gem-3.8	OpenCode	392	89.0	1.93	92.3	80.3	85.9	391	83.4	5.0	80.0
	Codex	392	92.6	2.01	97.0	83.6	89.8	391	84.3	2.5	82.5
	Claude Code	392	98.7	2.30	91.2	89.5	90.3	390	87.9	3.8	85.2
Qwen-3.8	OpenCode	392	96.2	2.61	79.9	90.1	84.7	391	90.1	4.7	86.7
	Codex	392	100.0	3.13	69.9	97.5	81.4	388	90.7	4.0	87.7
	Claude Code	392	98.7	2.53	84.9	92.7	88.6	389	84.8	4.0	82.0
DS-V4.1-F	OpenCode	392	95.9	2.36	85.1	88.1	86.5	390	87.6	5.2	83.9
	Codex	392	94.9	2.71	75.1	89.5	81.7	391	89.3	3.8	86.5
	Claude Code	392	100.0	2.89	75.3	95.2	84.1	390	88.7	3.9	86.0
GLM-5.3	OpenCode	392	95.9	2.77	75.0	89.0	81.4	391	88.1	5.7	84.1
	Codex	392	84.9	2.05	87.2	77.8	82.2	391	84.0	5.2	81.0
	Claude Code	392	100.0	2.94	73.3	96.6	83.4	388	85.2	6.6	80.8
Kimi-K3	OpenCode	392	88.5	2.15	87.4	80.8	84.0	391	86.2	6.2	82.1
	Codex	392	86.0	2.06	89.6	79.5	84.2	390	85.7	4.3	82.7
	Claude Code	392	96.2	2.33	85.5	86.2	85.9	391	81.5	7.2	76.7
HY3	OpenCode	392	93.9	2.17	87.4	82.3	84.8	388	81.4	7.3	76.8
	Codex	392	91.3	2.02	88.9	77.8	83.0	391	80.5	7.5	76.0
	Claude Code	392	97.4	2.21	89.0	86.4	87.7	388	79.8	8.2	75.0

M.2 Results by Change Category

Tables 18 to 20 report Ask-F1 and Plan Coverage for each change category of Table 6, one table per harness. In OpenCode, the lowest Plan Coverage for seven of the eight models occurs for performance and reliability changes or bug fixes, while no category is consistently the lowest in Ask-F1.

Table 18: Ask-F1 and Plan Coverage (%) by change category in OpenCode. Parentheses give the number of tasks in each category.

	Feature (249)		**Bug fix** (50)		**Compat.** (39)		**Data** (25)		**State** (17)		**Perf.** (12)	
Model	F1	Cov.	F1	Cov.	F1	Cov.	F1	Cov.	F1	Cov.	F1	Cov.
Opus-5	82.1	89.3	80.7	88.3	82.4	89.8	83.4	89.4	84.1	89.2	82.4	88.0
GPT-5.6-S	81.6	90.2	80.9	88.8	81.6	89.2	82.1	90.0	83.7	89.9	80.1	87.4
Gem-3.8	84.5	80.0	91.4	80.5	82.1	77.6	89.3	85.0	90.7	81.3	89.4	74.2
Qwen-3.8	84.0	86.6	85.3	83.6	85.3	86.8	89.1	91.8	84.6	88.3	86.5	87.2
DS-V4.1-F	86.2	84.7	85.9	78.3	89.3	85.1	85.1	88.7	88.4	82.7	88.9	79.7
GLM-5.3	82.0	84.3	79.8	82.1	81.4	81.6	81.0	85.6	78.9	87.8	79.4	86.6
Kimi-K3	83.9	83.0	80.5	74.1	87.0	81.1	88.8	90.1	84.5	83.2	79.2	81.6
HY3	85.0	76.7	84.0	76.9	86.7	77.6	85.2	84.4	83.3	71.9	78.3	68.0

Table 19: Ask-F1 and Plan Coverage (%) by change category in Codex. Parentheses give the number of tasks in each category.

	Feature (249)		**Bug fix** (50)		**Compat.** (39)		**Data** (25)		**State** (17)		**Perf.** (12)	
Model	F1	Cov.	F1	Cov.	F1	Cov.	F1	Cov.	F1	Cov.	F1	Cov.
Opus-5	82.8	84.9	79.0	83.7	82.4	84.7	83.8	85.1	83.1	84.9	81.1	84.3
GPT-5.6-S	77.8	87.5	83.4	86.3	75.3	87.3	64.3	87.7	73.4	87.5	70.5	86.9
Gem-3.8	89.2	81.7	92.4	83.3	87.7	82.6	92.4	86.9	92.4	84.0	89.1	84.2
Qwen-3.8	82.1	88.3	74.8	82.2	82.0	87.6	85.0	90.9	85.4	89.7	80.2	88.7
DS-V4.1-F	82.5	86.3	77.8	84.7	83.0	86.9	78.9	90.1	84.1	87.2	78.6	88.3
GLM-5.3	81.9	81.0	86.7	80.4	80.1	79.2	83.0	85.1	78.7	83.3	79.2	78.0
Kimi-K3	83.4	82.6	88.4	78.6	81.9	83.9	85.0	87.0	88.4	85.7	83.6	84.1
HY3	84.7	77.1	74.4	67.1	85.8	76.7	79.4	80.3	83.9	77.3	80.6	77.2

Table 20: Ask-F1 and Plan Coverage (%) by change category in Claude Code. Parentheses give the number of tasks in each category.

	Feature (249)		**Bug fix** (50)		**Compat.** (39)		**Data** (25)		**State** (17)		**Perf.** (12)	
Model	F1	Cov.	F1	Cov.	F1	Cov.	F1	Cov.	F1	Cov.	F1	Cov.
Opus-5	82.6	83.4	78.8	82.2	82.2	83.2	83.6	83.6	82.9	83.4	80.9	82.8
GPT-5.6-S	85.1	93.5	81.3	92.3	84.7	93.3	86.1	93.7	85.4	93.5	83.4	92.9
Gem-3.8	90.2	84.9	87.6	83.6	91.2	86.0	93.6	89.7	92.4	87.0	90.2	83.5
Qwen-3.8	89.0	81.7	86.3	78.2	87.9	84.4	88.9	85.9	91.1	84.4	88.0	84.0
DS-V4.1-F	84.5	86.2	82.8	81.0	82.3	87.1	85.8	90.1	85.5	87.2	81.3	87.9
GLM-5.3	83.7	81.1	82.4	75.6	82.8	81.9	83.4	86.0	84.1	81.7	83.0	80.3
Kimi-K3	85.6	77.0	89.7	72.5	86.0	75.9	83.0	80.2	86.1	79.6	82.5	79.8
HY3	87.2	74.6	91.0	76.1	86.1	72.2	86.3	78.2	91.0	79.4	87.7	75.4

M.3 Quantities behind the Observations

Tables 21 to 23 report the per-contrast and per-harness quantities summarized in the observations, computed over all eight models and 392 tasks. Answer incorporation is measured on a mechanism-enriched pilot of 60 cases and 131 plans, so its rates describe that pilot rather than all sessions. The coupling correlation is positive for every contrast, incorporation exceeds 98% in every harness, and Plan Coverage and Productive Precision vary far more across models than across harnesses.

Table 21: Coupling between clarification and planning for each harness contrast. The pooled correlation removes the mean of each contrast first.

Contrast	Pairs	$r(\Delta\text{Rec.}, \Delta\text{Cov.})$
OpenCode → Codex	3119	0.67
OpenCode → Claude Code	3116	0.59
Codex → Claude Code	3114	0.59
Pooled	9349	0.62

Table 22: Answer incorporation by harness (%). Exact and partial count answered blockers whose resolved value appears exactly or partially in the final plan, and contradiction counts those the plan contradicts.

Harness	Plans	Answered	Exact	Exact or partial	Contradiction
OpenCode	24	50	98.0	100.0	0.0
Codex	47	104	98.1	99.0	1.0
Claude Code	60	132	99.2	100.0	0.0

Table 23: Locus of variation for each metric. Harness columns give the mean over the eight models, the ranges are taken over model means and harness means, and ρ is the mean cross-harness Spearman correlation of model rankings, with tied values assigned their average rank.

Metric	OpenCode	Codex	Claude Code	Model range	Harness range	ρ
Plan Coverage	84.1	83.6	82.8	14.2	1.3	0.63
Productive Precision	83.2	84.1	82.4	20.7	1.7	0.83
Blocker Recall	85.3	83.2	90.5	19.1	7.3	0.65
Ask-F1	83.9	82.7	85.8	7.6	3.1	0.38
Ask Rate	92.1	90.4	97.6	18.6	7.2	0.69
Questions per Session	2.40	2.30	2.59	0.87	0.29	0.65

N Statistical Significance

All intervals in this section are 95% bootstrap percentile intervals from 10,000 paired task resamples, and a difference is significant when its interval excludes zero.

N.1 Paired Harness Comparisons

Table 24 reports the paired change of each metric when a model moves between harnesses, with the win, tie, and loss counts of Plan Coverage over tasks. Of the 24 model–contrast changes, 17 exclude zero in Plan Coverage. The coupling between changes in Blocker Recall and Plan Coverage has $r = 0.62$ [0.55, 0.70] pooled over contrasts after removing the mean of each contrast, and the correlation between the Productive Precision of a model, averaged over the three harnesses, and the harness-induced change in its Plan Coverage has $r = 0.47$ [0.08, 0.73] over the 24 model–contrast pairs.

Table 24: Paired harness comparisons within model and task. Values are target minus source in percentage points, with W/T/L the task-level wins, ties, and losses of Plan Coverage.

Model	ΔPrec.	ΔRec.	ΔCov.	W/T/L
OpenCode → Codex				
Opus-5	1.2 [−0.7, 2.3]	−1.2 [−3.6, 0.6]	−4.5 [−6.7, −3.1]	113/71/207
GPT-5.6-S	5.1 [3.8, 6.6]	−10.9 [−12.5, −8.7]	−2.5 [−4.0, −0.6]	120/102/168
Gem-3.8	4.7 [3.4, 5.4]	3.3 [1.7, 4.5]	2.5 [1.0, 3.4]	169/101/121
Qwen-3.8	−10.0 [−11.5, −8.1]	7.4 [5.6, 10.2]	1.0 [−0.7, 3.3]	141/124/123
DS-V4.1-F	−10.0 [−11.8, −8.8]	1.4 [−0.9, 3.3]	2.6 [0.5, 4.1]	170/100/120
GLM-5.3	12.2 [11.2, 13.6]	−11.2 [−12.4, −9.2]	−3.1 [−4.3, −1.5]	119/92/180
Kimi-K3	2.2 [0.7, 3.3]	−1.3 [−3.1, 0.3]	0.6 [−1.1, 1.9]	135/131/124
HY3	1.5 [0.4, 3.2]	−4.5 [−5.8, −2.2]	−0.8 [−2.1, 1.1]	123/128/137
OpenCode → Claude Code				
Opus-5	−0.7 [−2.7, 0.5]	1.5 [−1.0, 3.4]	−6.0 [−8.3, −4.5]	107/43/240
GPT-5.6-S	1.9 [0.5, 3.5]	3.9 [2.2, 6.4]	3.5 [1.9, 5.5]	187/86/117
Gem-3.8	−1.1 [−2.5, 0.1]	9.2 [7.4, 10.4]	5.2 [3.6, 6.2]	221/59/110
Qwen-3.8	5.0 [3.4, 7.0]	2.6 [0.7, 5.5]	−4.7 [−6.5, −2.3]	112/67/210
DS-V4.1-F	−9.8 [−11.7, −8.5]	7.1 [4.7, 9.1]	2.1 [0.1, 3.7]	161/107/122
GLM-5.3	−1.7 [−2.9, −0.3]	7.6 [6.3, 9.7]	−3.3 [−4.6, −1.6]	117/89/182
Kimi-K3	−1.9 [−3.5, −0.7]	5.4 [3.5, 7.1]	−5.4 [−7.2, −4.0]	109/57/225
HY3	1.6 [0.4, 3.4]	4.1 [2.7, 6.7]	−1.8 [−3.1, −0.1]	122/111/155
Codex → Claude Code				
Opus-5	−1.9 [−3.6, −0.8]	2.7 [0.6, 4.4]	−1.5 [−3.6, 0.1]	123/117/150
GPT-5.6-S	−3.2 [−4.4, −1.8]	14.8 [13.3, 16.9]	6.0 [4.5, 7.7]	237/47/106
Gem-3.8	−5.8 [−7.0, −5.2]	5.9 [4.4, 7.0]	2.7 [1.3, 3.5]	172/98/120
Qwen-3.8	15.0 [13.6, 16.8]	−4.8 [−6.6, −2.4]	−5.7 [−7.4, −3.6]	107/51/230
DS-V4.1-F	0.2 [−1.6, 1.4]	5.7 [3.5, 7.5]	−0.5 [−2.5, 0.9]	124/133/133
GLM-5.3	−13.9 [−14.8, −12.6]	18.8 [17.7, 20.7]	−0.2 [−1.3, 1.3]	124/137/127
Kimi-K3	−4.1 [−5.5, −3.1]	6.7 [5.0, 8.2]	−6.0 [−7.6, −4.8]	103/55/232
HY3	0.1 [−1.0, 1.6]	8.6 [7.4, 10.8]	−1.0 [−2.2, 0.8]	123/124/141

N.2 Model Differences

Table 25 compares adjacent models in the OpenCode ranking by Plan Coverage, paired over tasks. Five of the seven adjacent pairs are separated in Plan Coverage and five in Ask-F1, while the top two models, GPT-5.6-S and Opus-5, are separated in neither. The cross-harness rank correlations of Plan Coverage and Productive Precision are $\rho = 0.63$ [0.48, 0.77] and 0.83 [0.65, 0.86], and the verification gap has intervals [40.3, 45.0] for HY3 and [17.7, 21.0] for GPT-5.6-S.

Table 25: Paired differences between adjacent models in the OpenCode ranking by Plan Coverage (percentage points).

Pair	Models	ΔCov.	ΔAsk-F1
Rank 1 vs. 2	GPT-5.6-S vs. Opus-5	0.6 [−0.8, 1.7]	−0.5 [−1.9, 0.8]
Rank 2 vs. 3	Opus-5 vs. Qwen-3.8	2.5 [0.9, 3.8]	−2.6 [−4.3, −1.3]
Rank 3 vs. 4	Qwen-3.8 vs. GLM-5.3	2.6 [1.3, 3.7]	3.3 [2.0, 4.8]
Rank 4 vs. 5	GLM-5.3 vs. DS-V4.1-F	0.2 [−1.5, 1.6]	−5.1 [−6.8, −3.5]
Rank 5 vs. 6	DS-V4.1-F vs. Kimi-K3	1.8 [0.4, 3.0]	2.5 [1.2, 4.0]
Rank 6 vs. 7	Kimi-K3 vs. Gem-3.8	2.1 [0.6, 3.5]	−1.9 [−3.5, −0.6]
Rank 7 vs. 8	Gem-3.8 vs. HY3	3.2 [1.7, 4.5]	1.1 [−0.4, 2.4]

O External Coding Benchmarks

Figure 4a compares **`SWE-Plan`** with Terminal-Bench 4.0 (Merrill et al., 2026), the external coding benchmark with the widest exact-version coverage of the evaluated models. Table 26 lists its scores.

Four come from the official leaderboard (Terminal-Bench Team, 2026), and the scores of DS-V4.1-F and Kimi-K3, which the leaderboard does not list, come from the comparison table in the model card of DS-V4.1-F (DeepSeek-AI, 2026). Qwen-3.8 and HY3 have no exact-version result and are excluded. Because the agents behind these scores differ, Table 27 reports rank-based and linear fits over the six models, with exact permutation p-values and the range of Spearman ρ when each model is left out in turn. With six models, neither correlation reaches $p < 0.05$, but the sign of each holds in every leave-one-out subset, positive for Plan Coverage and negative for Ask-F1, which is the directional comparison made in Obs.❶.

Table 26: Terminal-Bench 4.0 resolution rates (%) of the evaluated models, with Plan Coverage and Ask-F1 in OpenCode.

Model	TB 4.0	Agent	Source	Plan Cov.	Ask-F1
Opus-5	51.82	Claude Code, max	Leaderboard	89.2	82.1
GLM-5.3	41.82	Claude Code, max	Leaderboard	84.1	81.4
GPT-5.6-S	37.27	Codex, max	Leaderboard	89.8	81.6
DS-V4.1-F	31.2	DeepSeek Harness Minimal	Model card	83.9	86.5
Gem-3.8	19.09	mini-SWE-agent, high	Leaderboard	80.0	85.9
Kimi-K3	12.6	n/d	Model card	82.1	84.0

Table 27: Fits of Plan Coverage and Ask-F1 to Terminal-Bench 4.0 over six models. Parentheses give exact permutation p-values, and the slope is in points per Terminal-Bench point.

Statistic	Plan Coverage	Ask-F1
Spearman ρ	0.77 (0.103)	-0.60 (0.242)
Kendall τ	0.60 (0.136)	-0.33 (0.469)
Pearson r	0.78 (0.068)	-0.62 (0.200)
OLS slope	0.21	-0.10
R^2	0.61	0.39
Leave-one-out ρ	[0.60, 0.90]	$[-0.60, -0.50]$

P Qualitative Cases

This section presents two cases selected from the recorded benchmark runs. The first follows one configuration whose clarification succeeds while its plan proposes verification that does not distinguish correct behavior from near misses. The second pairs the same model on the same task under two harnesses that differ in how much the model asks. We report only the questions, the replies of the User Simulator, excerpts of the frozen plans, and the judgments of the Plan Judge.

P.1 Case 1: Complete Clarification with Undistinguishing Verification

This case shows that an agent can recover every blocker and incorporate every answer while its plan still omits verification not requested by any input.

Case 1: setup

Model and Harness

Kimi K3 in the native Plan Mode of Claude Code.

Task

Gunicorn defines how leftover TLS request-body bytes are discarded after the application has finished reading, following a regression in which `ssl.SSLWantReadError` surfaces to application code. The request asks the agent to clarify the SSL-error handling scope and the terminal condition of the leftover drain, and to preserve ordinary non-TLS reads and successful full-body drains.

Registered Blockers

– **Handling scope.** Limit the handling to the `finish_body` body-draining path only.

– **Drain terminal condition.** Treat `SSLWantReadError` as a terminal end-of-body condition and stop draining immediately, even if some bytes were already read.

Verification Obligations

Three of the 11 obligations are verification obligations. A first-read test asserts that the error does not escape and that `body.read(8192)` is called exactly once. A partial-drain test returns two data chunks followed by the error and asserts exactly three reads with no retry. A normal-operation test asserts that an immediate empty read exits after one read.

Case 1: clarification and planning record

Questions and Matches

The agent asks two questions. "How should `ssl.SSLWantReadError` be handled, only when draining leftover request body in `Parser.finish_body`, or shared across all SSL read sites in the HTTP parser?" matches the handling-scope blocker. "When `ssl.SSLWantReadError` is encountered while draining leftover body bytes in `finish_body`, what should the terminal condition be?" matches the terminal-condition blocker. Both questions are productive, and the User Simulator returns both registered answers.

Plan

The plan confines the handler to `finish_body`, wraps the drain loop in `try` with `except ssl.SSLWantReadError: pass`, and states "No retry, no select, no time.sleep, no bounded-attempt loop". Its verification section proposes a test whose first assertion is "Assert: no exception propagates", a test over an unreader that raises the error after delivering a partial body, and a companion test in which the remaining bytes are consumed.

Judge Outcome

All eight obligations on behavior, condition, ownership, and preservation are supported. All three verification obligations are absent. The Plan Judge notes that the first test "does not commit to asserting `body.read(8192)` is called exactly once, so a retrying implementation would not be distinguished", that no test returns two data chunks before the error and asserts exactly three reads, and that no test asserts a clean exit after an immediate empty read. Productive Precision and Blocker Recall are both 1.0, and Plan Coverage is 72.7.

Table 28 shows that the pattern is not specific to this configuration. On the same task, every configuration under Claude Code and Codex recovers both blockers, and fourteen of the sixteen leave all three verification obligations absent.

Table 28: Case 1 across configurations. Recall denotes Blocker Recall, Other denotes the supported share of the eight non-verification obligations with conflicting obligations in parentheses, and V1 to V3 denote the judgments of the three verification obligations.

Harness	Model	Recall	Other	V1	V2	V3	Cov.
Claude Code	Claude Opus 5	1.0	8/8	A	A	A	72.7
Claude Code	DeepSeek V4.1 Flash	1.0	8/8	S	A	A	81.8
Claude Code	Gemini 3.8 Flash	1.0	8/8	A	A	A	72.7
Claude Code	GLM 5.3	1.0	8/8	A	A	A	72.7
Claude Code	GPT-5.6 Sol	1.0	8/8	S	A	A	81.8
Claude Code	HY3	1.0	7/8 (1 C)	A	A	A	57.9
Claude Code	Kimi K3	1.0	8/8	A	A	A	72.7
Claude Code	Qwen3.8 Max	1.0	8/8	A	A	A	72.7
Codex	Claude Opus 5	1.0	8/8	A	A	A	72.7
Codex	DeepSeek V4.1 Flash	1.0	8/8	A	A	A	72.7
Codex	Gemini 3.8 Flash	1.0	8/8	A	A	A	72.7
Codex	GLM 5.3	1.0	8/8	A	A	A	72.7
Codex	GPT-5.6 Sol	1.0	8/8	A	A	A	72.7
Codex	HY3	1.0	8/8	A	A	A	72.7
Codex	Kimi K3	1.0	8/8	A	A	A	72.7
Codex	Qwen3.8 Max	1.0	7/8 (1 C)	A	A	A	57.9

➠ **Explicit information.** Both blockers are asked, both answers enter the plan, and every obligation stated or implied by the request and the answers is supported.

- **Implicit verification.** The plan contains tests, but none of them distinguishes the correct drain from its near misses, so an implementation that retries after the error would pass them.
- **Interpretation.** Clarification succeeds completely in this task, yet the resulting plans stop at what the inputs state. The missing obligations are exactly those that no request or answer asks for, which is the shared deficit in implicit requirements that joint measurement of the two dimensions exposes.

P.2 Case 2: Higher Willingness without Higher Precision

This case pairs the same model on the same task under two harnesses. The harness that elicits more questions does not elicit more matched questions, and the plan settles the unmatched points by assumption.

Case 2: paired setup

Model and Harnesses

GLM 5.3 in the native Plan Mode of Codex and of Claude Code.

Task

Conda defines the user-visible error when sharded repodata is disabled and the classic `noarch/repodata.json` of a channel subdir returns 404, including whether the failure should mention enabling shards. The request asks the agent to clarify the shards-only guidance surface and the probe-failure fallback, and to keep the existing behavior for ordinary missing channels.

Registered Blockers

- **Guidance surface.** Attach a structured guidance payload to `UnavailableInvalidChannel` through a new optional `shards_only_hint` parameter.
- **Probe-failure fallback.** Treat a failing shard-index probe as an absent index, so the original unavailable-channel error is raised without the hint.

Held Constant

The model, task, repository revision, User Simulator, Plan Judge, and the checklist of 14 obligations are identical across the two runs. Among the obligations, the probe issues a `HEAD` request to the shard-index resource, and any 2xx response counts as existence.

Codex run

Questions and Matches

The agent asks two questions. "When shards are disabled and classic `noarch/repodata.json` is missing, where should guidance to enable shards appear?" matches the guidance-surface blocker. "What should happen when the opportunistic shard-index probe fails?" matches the probe-failure blocker. The User Simulator returns both registered answers.

Judge Outcome

Thirteen obligations are supported, including the probe and existence obligations, and one preservation obligation on the timeout and proxy settings of the probe is A. No obligation is in conflict. Productive Precision is 1.0, Blocker Recall is 1.0, and Plan Coverage is 92.9.

Claude Code run

Questions and Matches

The agent asks four questions. "If the opportunistic shard-index probe itself fails, what should the user see?" matches the probe-failure blocker, and "How should the enable-shards mention be presented to the user?" matches the guidance-surface blocker. "When should the resulting `UnavailableInvalidChannel` error mention enabling sharded repodata?" and "When should the probe-and-hint behavior be active?" match no blocker and receive the fixed template reply, "I don't have additional information for that. Please proceed using the repository context and state any assumptions in the plan."

Plan

For the two unmatched points, the plan adopts the option that the agent recommended in its first question: mention shards "only if that index actually exists (200)". It probes the index with `get_session(subdir_url).get(..., stream=True)` and returns `response.status_code == 200`.

Judge Outcome

Twelve obligations are supported, and two are in conflict. The plan commits to a `GET` request although the obligation requires a `HEAD` request, and it treats only status 200 as existence although the obligation accepts any 2xx response, which is the near miss recorded for that obligation. Productive Precision is 0.50, Blocker Recall is 1.0, and Plan Coverage is 73.5.

- **Controlled difference.** Only the harness changes between the two runs.
- **Codex outcome.** Two questions match both blockers, and the plan contains no conflicting commitment.
- **Claude Code outcome.** Four questions match the same two blockers, and the two additional questions obtain no information. The plan settles those points by the agent's own assumption and conflicts with two obligations.
- **Interpretation.** The harness doubles the number of questions without raising Blocker Recall, so the additional willingness to ask brings no additional information. The resulting plan is worse rather than better, because its commitments on the unmatched points rest on assumptions. A harness can raise how much a model asks, but the plan improves only when the questions are accurate.