REQAUDITBENCH: BENCHMARKING CODING AGENTS ON PROACTIVE DISCOVERY OF REQUIREMENT–CODE INCONSISTENCY

Anonymous authors
Paper under double-blind review

ABSTRACT

Coding agents increasingly write code for software projects, but checking whether the code is consistent with documentation is left to human reviewers. Existing benchmarks do not test whether agents can take over this requirement–code consistency audit. Implementation benchmarks do the check for the agent with prepared tests, and consistency checkers are given the function to examine. In a real repository, the documentation states many requirements and marks none as broken, so the auditor must first decide which promise to check. We introduce ReqAuditBench to measure this proactive investigation. Each task gives an agent a repository and a region of its documentation that contains at least one violated requirement, without saying which one. We build tasks from pull requests that update both a documented requirement and its implementation. We place the updated documentation into the old repository, where nothing marks which passage changed. On 109 tasks with 134 requirements from 9 Python repositories, eight coding agents reach only 35.1–57.5% requirement recall, the share of violated requirements they report. Seeing a requirement is often not enough. For 60.3–92.6% of missed requirements, the promise had already appeared in the agent's tool output. In case studies, agents stop before checking the code behind the promise. The results suggest that coding agents are not yet ready to take over this audit.

1 INTRODUCTION

Coding agents are increasingly used to change code in real repositories. Current benchmarks give them a reported issue or a requested feature and evaluate the code they write (Jimenez et al., 2024; Zhou et al., 2026). The development workflow behind these tasks has three stages. A developer first writes the requirements, the agent then writes the code, and someone finally checks that the code does what the requirements say. We call this last stage the *requirement–code consistency audit*. Agents have taken over the second stage, but studies of agent-written code show that the third stage still falls on human reviewers (Asdaque et al., 2026).

The requirements that this audit checks against do not vanish after a task closes. They accumulate in the repository's documentation pages and API docstrings, which record accepted inputs, configuration options, and expected behavior. We call each such documented promise about externally visible behavior a *requirement*. Such documentation is written in two ways. Some is written from existing code, by hand or with summarization tools (Sridhara et al., 2010; Luo et al., 2024), so it matches the code when it is written. Other documentation states the intended behavior and is updated in the same change as the code, as when a pull request edits a docstring to announce a new default (Monperrus et al., 2012; Fluri et al., 2007; Tan et al., 2007). Nothing forces the code to keep such a promise. Keeping code consistent with these promises is a long-standing maintenance problem (Aghajani et al., 2019; Wen et al., 2019; Panthaplackel et al., 2021), and inconsistent changes are about 1.5 times as likely as consistent ones to introduce a bug (Tan et al., 2007; Radmanesh et al., 2024).

We therefore ask whether agents can take over this audit as well, while humans continue to write the requirements (Figure 1(a)). Current benchmarks do not measure this ability. Implementation benchmarks perform the check themselves with prepared fail-to-pass tests, so the agent is never

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

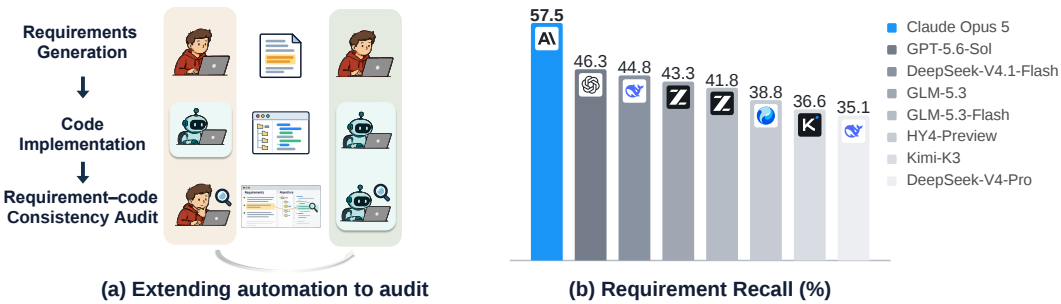


Figure 1: (a) Software development has three stages: writing requirements, implementing code, and auditing whether the code keeps them. Agents now implement, and ReqAuditBench measures whether they can also audit, which existing benchmarks do not test. (b) Eight coding agents recover only 35.1–57.5% of violated requirements.

asked whether the code keeps a requirement (Jimenez et al., 2024; Zhou et al., 2026). Prior work on documentation–code inconsistency does ask a model to check, but only in a *local audit*, where the method to check is given together with its documentation (Lee et al., 2025; Kiecker et al., 2026). A maintained repository instead needs a *global audit* of its whole implementation against its documentation. Checking each change locally is not enough, because a change to shared code can break a promise documented for another function, and many repositories already carry inconsistencies left by earlier changes (Wen et al., 2019; Tan et al., 2023). The documentation records many promises and does not say which one the code breaks. The auditor must decide which promises to examine, trace each to its implementation, and substantiate the violation. An agent told where to look, as in an implementation task or a local audit, does not show that it can make this decision.

Therefore, we introduce **ReqAuditBench**, a benchmark that evaluates whether agents can perform this global audit, that is, proactively discover documented requirement violations in a repository. Specifically, each task gives the agent one version of a repository to investigate inside a container, together with a region of its own documentation that contains at least one violated requirement. The region is either a chapter of documentation pages or the docstrings of one code module. The agent sees no diff or bug fix to compare against, and it may inspect implementation anywhere in the repository. The agent must report which requirements the implementation fails to keep, quoting the promise and describing the contradictory behavior. No repair is required. Crucially, the task is a global audit rather than a local one. The agent is not told which promise is violated or where it is implemented, and must decide both on its own.

We collect candidates from maintenance changes that update both documentation and implementation. This raises two benchmark-design challenges. **First, co-changes require evidence of a violation.** They can reflect refactoring or clarification. Establishing a target requires showing that the preceding implementation violates the updated promise under its stated conditions. **Second, discovery requires preserving evidence without revealing the target.** Maintenance records help establish answers but also reveal targets. Isolating the target passage turns discovery into directed checking, while removing context can obscure the promise's conditions.

ReqAuditBench addresses these challenges with two design choices. **1) Evidence-linked requirements.** We use the full pull-request diff to link each updated promise and its conditions to contradictory pre-change behavior, excluding unsupported candidates. **2) Documentation write-back.** We write the updated documentation pages and docstrings back into the pre-change repository in full, not as a diff, and leave the implementation unchanged. The updated promise therefore sits among unchanged ones with nothing marking it. The reference card, which records the violated requirements for scoring, stays hidden together with repair records (Section 2.3).

We evaluate eight models as coding agents on 109 tasks and 134 requirements (Section 3). GPT-5.6-Sol runs in Codex CLI and the other seven models run in Claude Code. Requirement recall ranges from 35.1% to 57.5%. Even the best model, Opus 5, recovers only 77 of the 134 requirements, fewer than three in five. The recorded trajectories show that agents fall short in both steps of an audit. The first step is selecting a promise for investigation, that is, choosing which promise to check. Missing

a requirement is often not a matter of never seeing it. For 60.3–92.6% of the requirements an agent misses, the text of the promise had already appeared in its tool output. Nor do the agents with higher recall read more of the documentation: Opus 5 and GPT-5.6-Sol open the smallest share of the supplied documentation files. In selected scikit-learn Array API cases, Opus 5 and GPT-5.6-Sol both read the documentation entry of the target, but only Opus 5's recorded trajectory follows it into the implementation and its tests and reports the violation (Section 4.1). The second step is verifying the violation, that is, checking the chosen promise against the actual behavior. Agents, Opus 5 among them, flag a candidate but leave it unchecked (Section 4.2). For recurring audits, recall also trades against recorded duration and estimated token fees (Figure 3): Opus 5 offers the highest recall at the highest mean estimated token fees and nearly twice GPT-5.6-Sol's recorded duration, whereas GLM-5.3-Flash gives up 4.5 percentage points of recall for 89.0% lower mean token fees but takes longer, 28.6 versus 13.7 minutes (Section 3.3). Token fees are estimated from public prices for the recorded runs.

Our contributions are summarized as follows:

- We introduce ReqAuditBench, a benchmark that evaluates whether coding agents can check a repository's code against its own documentation without being told which requirement is broken. It contains 109 tasks with 134 violated requirements from 9 Python repositories.
- We propose a method that builds such tasks from real pull requests that change both documentation and code. The pull request confirms that the old code breaks the updated documentation, and the updated documentation is placed into the old repository, where nothing marks which passage changed.
- We evaluate eight coding agents and analyze their trajectories. The best agent recovers only 57.5% of the violated requirements. Most requirements an agent misses had already appeared in its tool output, and case studies show agents failing to choose a seen requirement for checking or to verify it against the code.

2 REQAUDITBENCH

ReqAuditBench evaluates whether a coding agent can audit a repository against its own documentation when no one says which requirement is broken. It comprises 109 tasks with 134 violated requirements, each built from a merged pull request in one of 9 Python repositories. Section 2.1 defines the task, Section 2.2 the metrics, Section 2.3 the construction, and Section 2.4 the composition of the benchmark.

2.1 TASK FORMULATION

A *requirement* is a documented promise about externally visible behavior. Each *task* is built from one merged pull request and consists of a repository, a documentation region, and a reference card that only the evaluator sees. The repository is the project as it stood before the pull request, with the updated documentation written back (Section 2.3). We call this version of the project the task *snapshot*. The region contains at least one violated requirement, and the prompt does not say which one. Depending on where the task's requirements are documented, the region is a *page region*, a set of documentation pages from one chapter of the documentation's table of contents, or a *docstring region*, the set of Python files in one code module, whose docstrings serve as the documentation. Appendix A gives the region-building rules. The agent may inspect implementation anywhere in the repository and reports at most five findings. Each finding quotes a promise from the region and describes the implementation behavior that contradicts it. No repair is required. The reference card lists the task's violated requirements.

2.2 EVALUATION METRICS

Each requirement on the reference card consists of one or more *behaviors*. A behavior is one checkable case of the promise, given by an object, the documented condition, the promised outcome, and the contradicting pre-change outcome. A finding that identifies any behavior of a requirement recovers that requirement. Findings are matched in two steps. A mechanical check first discards

findings that lack an actual-behavior description or whose quotation does not appear verbatim, ignoring whitespace, in the cited region file. It credits a finding directly when its quotation overlaps the evidence of exactly one violated behavior. For the main results, three LLM judges from different providers, Claude Fable 5, GPT-6-Sol, and Qwen3.8-Max, then judge the remaining findings independently. A finding matches a behavior when at least two judges select that same behavior. A finding whose quotation overlaps the evidence of several violated behaviors, and of no other behavior, is already credited, and the judges only decide which of these behaviors it identifies. Matches are grouped by requirement, with at most one hit per requirement. Appendix C gives the full rules, judge agreement, the judge prompts, and a quotation-overlap comparison.

Requirement recall (RR) measures the fraction of registered violated requirements identified by an agent's findings. Let $\mathcal{T}$ be the set of tasks, $\mathcal{R}_t$ the registered requirements of task t, $\mathcal{R} = \bigcup_t \mathcal{R}_t$, and $\mathcal{H} \subseteq \mathcal{R}$ the requirements recovered by an agent's findings, each counted at most once. We report

$$\mathrm{RR} = \frac{|\mathcal{H}|}{|\mathcal{R}|}, \qquad \mathrm{THR} = \frac{\left|\{\, t \in \mathcal{T} :\ \mathcal{H} \cap \mathcal{R}_t \neq \emptyset \,\}\right|}{|\mathcal{T}|},$$

with $|\mathcal{T}| = 109$ and $|\mathcal{R}| = 134$. Both metrics use the full benchmark denominators. RR pools requirements across tasks rather than averaging per-task rates, whereas THR counts tasks with at least one recovered requirement.

2.3 Benchmark Construction

A maintenance change can reveal both the intended behavior and the code needed to implement it, but giving that change to an agent would also disclose what to investigate. Construction therefore proceeds in three stages (Figure 2). Candidate mining finds changes that may carry a violated requirement. The reference-requirements stage establishes, for the evaluator, which requirements the pre-change code violates. Target-hidden task construction then rebuilds the evidence the agent may inspect without the record that reveals them. Each stage checks its own output before the next stage consumes it.

Stage 1: Candidate mining. We selected nine actively maintained Python repositories from three domains: data science (scikit-learn, xarray, pandas), machine and deep learning (Transformers, vLLM, slime, MLflow), and general-purpose infrastructure (FastAPI, pytest). Many users of data-science and machine-learning libraries rely on documentation that states behavior densely, and the two infrastructure libraries serve as a contrast. Appendix A.1 gives the selection criteria. From their merged pull requests, we keep those that change both the implementation and the user-facing documentation of an existing public object, whether as a fix or a new feature. Documentation means pages and public docstrings, not changelogs or comments. A pull request can also contain unrelated edits, so this stage yields candidates, not evidence of a violation.

Stage 2: Reference requirements. For each candidate, Claude Fable 5 drafts a reference card. Two reviewers, each with more than three years of software development experience, check every card independently, and a third reviewer with the same experience resolves disagreements. Of 243 candidates with drafted cards, 109 remain as tasks after review. Requirements are fixed from the pull-request description before any code is read. The full diff is then read to pair each changed documentation block with the implementation change, together with the pre-change code, including pre-change files outside the diff when a dependency is needed. A behavior becomes a target only if the pre-change implementation contradicts the promise under the documented conditions. Changes to code or tests alone are insufficient. The card records the public object, applicability conditions, and promised and pre-change behavior. Already satisfied behaviors are excluded, and behaviors that the evidence leaves unsettled are not scored. Repeated passages are grouped under one requirement, the scoring unit. Appendix A details the review protocol.

Stage 3: Target-hidden task construction. The reference identifies what is wrong, but the agent must discover it from the repository. Supplying only the target quotation would turn discovery into directed checking, whereas using the post-change repository would import the implementation that satisfies it. We instead start from the pre-change repository and write back complete changed documentation pages, docstrings of existing objects, and navigation entries for new pages, leaving

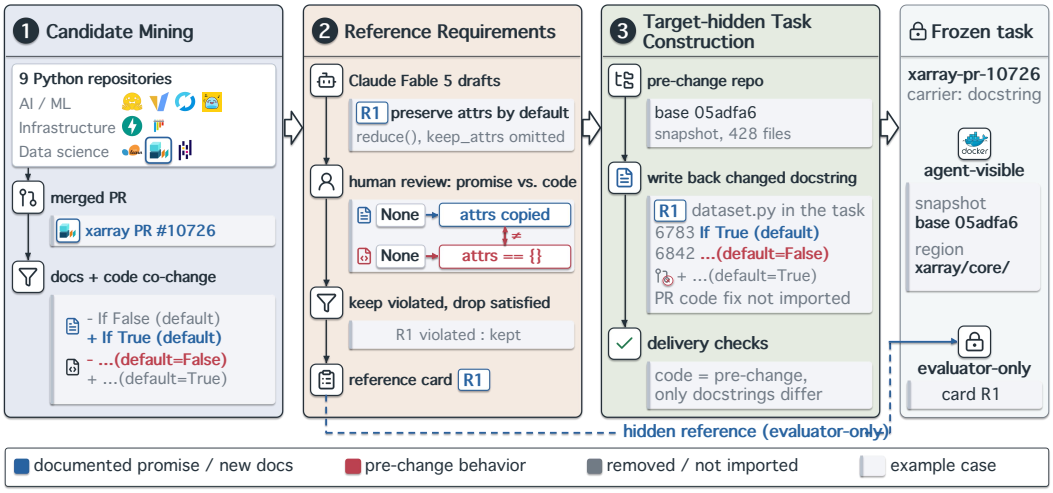


Figure 2: Benchmark construction pipeline, illustrated with xarray PR #10726. (1) Candidate mining keeps pull requests changing both documentation and implementation. (2) Reference requirements: Claude Fable 5 drafts and humans review violated requirements. (3) Target-hidden task construction writes changed documentation into the pre-change repository, hiding the reference card.

implementation and tests unchanged. Docstrings are replaced within their existing objects rather than by copying the updated source files. Changelogs and new-object docstrings are excluded. The reference card, patch, and version-control history remain hidden, and the agent receives the page or docstring region of Section 2.1, with no target passage highlighted. Consistency checks then tie the task back to Stage 2: the implementation must match the pre-change code, modified Python files must keep identical syntax after docstrings are stripped, and every reference quotation must occur in the snapshot. Task identities, targets, documentation types, source links, and check records are then fixed together.

2.4 Benchmark Composition

The benchmark contains 109 tasks and 134 requirements across 9 Python repositories. Most tasks have one violated requirement, but some have several. A **page** requirement is stated on a documentation page, whereas a **docstring** requirement appears only in a Python object's docstring or attribute documentation. The benchmark contains 69 page tasks and 40 docstring tasks, with 83 and 51 requirements, respectively. Docstrings can expose their containing module and thus provide different localization cues from pages. These groups broaden the documentation represented but differ in repositories and requirements, so they are not a controlled comparison of presentation formats.

Tasks come from four kinds of pull requests: new features (48 tasks, 62 requirements), bug fixes (28, 31), deprecations or removals (15, 17), and other intended behavior changes such as a new default or new validation (18, 24). New-feature tasks model code that has not yet delivered a written promise, and fix tasks model code that breaks one it should already keep. An audit must catch both. In 51 tasks the pre-change code lacks the documented capability, and in 58 it has it but behaves differently. Claude Fable 5 drafted these labels and the authors checked them (Appendix D.2).

Table 1 contrasts this setting with prior benchmarks (Jimenez et al., 2024; Zhou et al., 2026; Liu et al., 2026; Lee et al., 2025; Kiecker et al., 2026). Each prior task centers on one named target. Repair, feature, and test-generation benchmarks give the agent an issue, a feature description with its interface, or an entry point, and method-level checkers pair one documented method or function with its implementation. ReqAuditBench instead gives an open documentation region drawn from maintainer-written pages, which belong to no single method, and docstrings. The region states many requirements and marks none as violated, so the agent must decide which ones to check. It reports findings rather than a patch or tests.

Table 1: Why a new benchmark is needed. Each prior task names its target: an issue, a feature, an entry point, or a documented method. ReqAuditBench gives only a documentation region, so the agent must decide what to check. The comparison concerns inputs and outputs, not difficulty.

Work	Target of a task	Agent is given	Output	Scored against
Write code or tests for a named target				
SWE-bench	One issue	the issue description	Patch	Tests
FeatureBench	One feature	a feature description, interface	Module	
TestExplora	One entry point	the entry point	Tests	
Judge a named documentation–code pair				
METAMON	One method	the method and its documentation	Verdict	Annotated labels
CASCADE	One function	the function and its documentation		
ReqAuditBench	**Open region**	maintainer-written docs, no target	Findings	Reference requirements

3 EXPERIMENTS

We evaluate how well the eight coding agents recover violated requirements (Section 3.2) and how recall trades off against recorded duration and estimated token fees (Section 3.3). Section 4 then examines the investigation processes behind selected outcomes.

3.1 EXPERIMENTAL SETUP

Coding agents. We compare eight coding agents, each a model running in an agent harness. GPT-5.6-Sol uses Codex CLI, whereas the other seven use Claude Code. All request high effort, with full settings in Appendix B. The results compare complete agents, model plus harness, rather than isolated model weights.

Runtime and prompt. All agents receive the same instructions and documentation regions, with a five-finding cap. Each task provides a read-only snapshot, a writable work directory, a two-hour budget, and a ten-minute finishing allowance. The setting is a source-level audit rather than a pre-configured application deployment: a Python runtime supports commands and local scripts, but project-specific dependencies are neither preinstalled nor installable. Agents can inspect source when a check requires those dependencies. The prompt requests counterexamples and incremental reports. The container is cut off from external services such as GitHub and PyPI, and snapshots carry no version history. Appendix B.2 gives the execution interface.

3.2 OVERALL PERFORMANCE

Even the strongest agent recovers only 57.5% of the violated requirements. Claude Opus 5 leads with 77/134 requirements, followed by GPT-5.6-Sol at 62/134 (46.3%). Across agents, recall ranges from 35.1% to 57.5%. Even the leader misses 57 requirements despite being given the relevant documentation region and told that it contains an inconsistency. A model-free lookup baseline recovers 4 (3.0%, Appendix D.3). Opus 5's lead over GPT-5.6-Sol comes mainly from new-feature requirements (18 recovered only by Opus 5, 6 only by GPT-5.6-Sol). On bug fixes, GPT-5.6-Sol recovers 17 of 31 to Opus 5's 15 (Table 8). Figure 1(b) compares requirement recall, and Table 2 also reports task hit rate, the agent harnesses, and mean input/output tokens. Scores use all 109 tasks and 134 requirements, and Appendix F defines token accounting.

3.3 PERFORMANCE–COST TRADE-OFFS

No single agent leads on recall, recorded duration, and token fees at once. In practice, choosing an auditing agent also depends on how long developers can wait and how often an organization can afford to run it. Figure 3 relates requirement recall to mean model responses, recorded duration, output tokens, and estimated token fees per recorded run. The observations suggest three deployment priorities.

Table 2: Results on 109 tasks and 134 requirements. Even the best agent recovers only 57.5% of requirements. RR is requirement recall and THR is task hit rate, the share of tasks with at least one recovered requirement. In each column, bold is best and underlined is second best. For tokens, fewer is better. Input includes cache tokens.

Agent harness	Model	Metrics (%)		Mean tokens/run	
		RR	THR	Input (M)	Output (k)
Claude Code	Claude Opus 5	**57.5**	**59.6**	21.10	164.3
Codex CLI	GPT-5.6-Sol	<u>46.3</u>	<u>48.6</u>	**7.70**	**32.9**
Claude Code	DeepSeek-V4.1-Flash	44.8	46.8	30.68	211.0
Claude Code	GLM-5.3	43.3	47.7	19.76	157.0
Claude Code	GLM-5.3-Flash	41.8	45.9	<u>13.02</u>	<u>95.4</u>
Claude Code	HY4-Preview	38.8	44.0	33.07	243.9
Claude Code	Kimi-K3	36.6	41.3	20.62	149.4
Claude Code	DeepSeek-V4-Pro	35.1	37.6	13.24	107.0

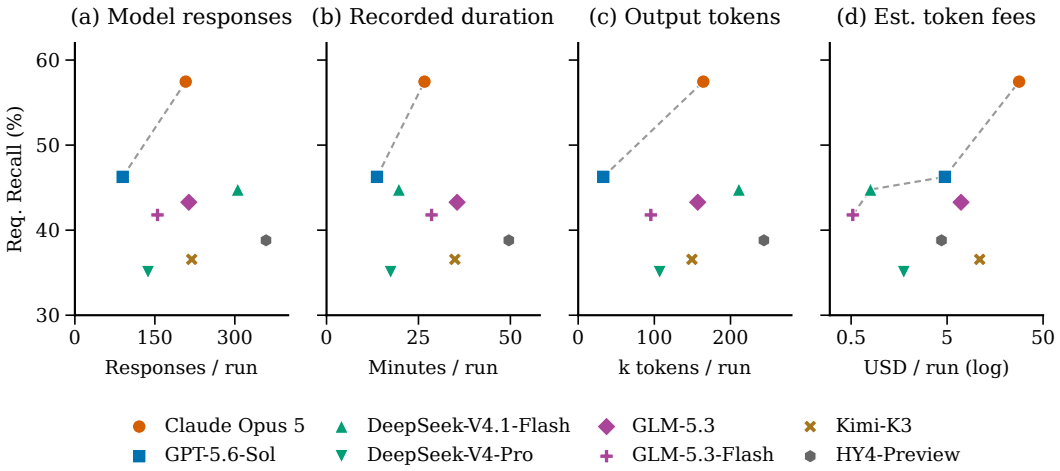


Figure 3: No agent leads on recall, speed, and cost at once. Each point is one agent's recall against its mean resource use per run. Dashed lines join agents no other agent beats on both. Claude Opus 5 has the highest recall at the highest fee, GPT-5.6-Sol is fastest with the second-highest recall, and GLM-5.3-Flash is cheapest. Panel (d) is log-scaled.

For coverage-first audits, Opus 5 gives the highest recall at the highest token fee. Its requirement recall is 57.5%, and its estimated token fee is $28.13 per run. Its mean recorded duration is 26.6 minutes, nearly twice GPT-5.6-Sol's 13.7 minutes. It is therefore a candidate when recovering more violations takes priority over recorded duration and token fees.

For fast feedback, GPT-5.6-Sol finishes fastest while keeping the second-highest recall. It uses the fewest model responses (90 on average) and has the shortest mean recorded duration among the eight agents, while retaining 46.3% recall. Its $4.75 mean token fee, however, is materially higher than GLM-5.3-Flash's. Recorded duration reflects the full agents, including their different harnesses.

For budget-limited recurring audits, GLM-5.3-Flash costs 89.0% less than GPT-5.6-Sol for 4.5 points of recall. It has the lowest mean token fee ($0.52 per run) and reaches 41.8% recall. This trade-off is relevant to frequent or offline audits where token fees limit how often a codebase can be checked. The saving comes with a longer recorded duration: GLM-5.3-Flash takes 28.6 minutes on average, versus GPT-5.6-Sol's 13.7.

4 Analysis of the Audit Process

Turning a documented promise into a reported violation takes two steps: selecting a promise for investigation, and verifying its violation. To examine the recall gap in Section 3.2, we trace how documented promises become reported violations. We combine counts computed mechanically over all recorded trajectories with selected trajectories whose steps we checked against the repository source. Together these reveal both steps (Sections 4.1 and 4.2).

4.1 Selecting a Promise for Investigation

Most missed requirements had already appeared in the agent's tool output. Across agents, 60.3–92.6% of requirements missed in completed runs have a reference quotation in a tool output (Appendix E.3), motivating a closer look at how agents use the available promise.

Reading more of the region does not explain the recall gap. On the 76 tasks with a recorded run from every agent, Opus 5 and GPT-5.6-Sol have opened 18–21% of the supplied region files by turn 10, compared with 35–52% for the other six agents, and end with the lowest coverage, 61% versus 62–75% (Figure 4). Across the eight agents, coverage at turn 10 is inversely related to recall (Spearman $\rho = -0.79$). Opus 5 and GPT-5.6-Sol open the region at almost the same pace yet differ by 11 points in recall, so what separates them lies in how they use the promises they read.

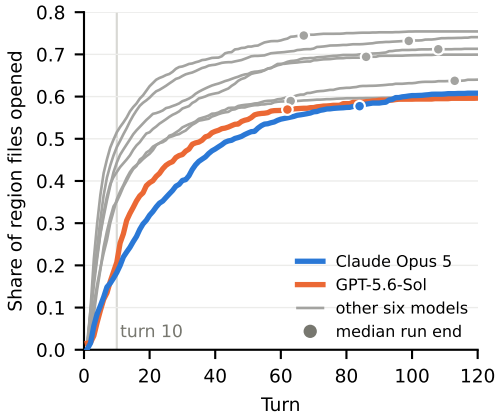


Figure 4: Reading more of the region does not raise recall. Each curve shows the share of region files an agent has opened by each turn, on the 76 tasks all agents ran. The two highest-recall agents open it most slowly. Dots mark median run ends (Appendix E.4).

In five paired cases where both agents saw the same promise, Opus 5 followed it into the implementation, whereas GPT-5.6-Sol turned to other promises. We inspect all five requirements recovered by Opus 5 but missed by GPT-5.6-Sol within the nine-task scikit-learn Array API group. In every pair, both agents see the target's entry in the documented list of supported functions. Opus 5 follows these claims into target function bodies, namespace handling, and tests. In PR 32597, it traces `manhattan_distances` to `distance.cdist` and checks the wrapper's dispatch, whereas GPT-5.6-Sol investigates shared utilities and `mean_squared_error`, reporting other promises. These outcome-selected cases show that seeing a promise does not by itself lead an agent to check it. Appendix E records the individual checks.

4.2 Verifying the Violation

Flagging the right candidate is not enough if the agent stops before checking the code path that decides it. Once a promise is selected, the investigation must resolve whether the implementation fulfills it. In scikit-learn PR 31671, DeepSeek-V4.1-Flash flags two d2 score functions but assumes a helper "uses xp presumably." Its later scripted keyword search again raises the targets without checking that delegated path, leaving them unreported. By contrast, GLM-5.3-Flash checks `np.exp(K, K)` in `laplacian_kernel` against namespace-aware neighboring kernels and reports the mismatch in PR 32613. This provides a successful verification example from another agent.

The leading agent also leaves checks unfinished. In PR 34785, Opus 5 reads `contingency_matrix`, identifies NumPy/SciPy-specific operations, and notes that it "needs more digging," but omits the requirement. All eight agents miss this case.

5 Related Work

Coding-agent benchmarks. Coding-agent benchmarks usually specify the change to make or review. SWE-bench gives an issue and scores the patch with the tests tied to it, and FeatureBench also gives the interface of the function to implement (Jimenez et al., 2024; Zhou et al., 2026). SWE-PRBench and c-CRAB give a pull request to review and score the review against the comments of its human reviewers, which c-CRAB turns into executable tests (Kumar, 2026; Zhang et al., 2026). Review is the closest of these tasks to an audit, but the pull request already marks the code to examine. Other evaluations check whether agent patches follow design constraints or keep maintainable structure (Yu et al., 2026; Zhu et al., 2026). In each case the change tells the agent what to examine, and the agent is judged on how well it handles that change. ReqAuditBench gives no change, so the agent must decide for itself what needs attention.

Documentation–code consistency. Other work checks code against its documentation. METAMON records the current behavior of a documented method as regression tests and asks an LLM whether each assertion agrees with the documentation (Lee et al., 2025). CASCADE generates tests from the documentation of a function and reports an inconsistency only when the original code fails them and an implementation generated from the same documentation passes (Kiecker et al., 2026). TRACE perturbs the documentation, the implementation, or both in single Java methods, and finds that models detect the mismatch less often when only the implementation changes (Ulfat et al., 2026). Earlier methods flag comments that a code change makes outdated, classify method–comment pairs, or flag references to removed code elements (Panthaplackel et al., 2021; Dau et al., 2024; Zhong et al., 2025; Tan et al., 2022; 2023). Each check starts from a method, a comment, or a change already paired with its documentation, whereas ReqAuditBench leaves that pairing to the agent, which must find both the promise and the code that decides it.

Proactive auditing. A third line of work asks agents to find defects that no one has reported. These benchmarks remove the issue report, but the known fix still defines the defect and often where to look. Active-SWE asks agents to find and fix several bugs in a repository without an issue report. It supplies files extracted from the reference patch and scores the repair with the reference tests (Li et al., 2026). TestExplora names the entry functions that the reference tests call, documents them from the repaired code, and scores generated tests that fail before the fix and pass after it (Liu et al., 2026). It also excludes pull requests that change documentation, the source from which ReqAuditBench builds its tasks. SciCoQA gives no target location and asks for discrepancies between a scientific paper and its code, but the promises come from the paper rather than from the repository (Baumgärtner & Gurevych, 2026). ReqAuditBench differs on three points. The defect is defined by a promise that maintainers wrote in the repository's own documentation, not by a fix or a separate paper. The agent receives the repository and a documentation region, but no files from the patch, no entry point, and no target location. It reports which promise the code breaks, quoting the promise and describing the contradicting behavior, and is scored against human-reviewed reference requirements rather than tests from the fix.

6 Conclusion

ReqAuditBench tests whether coding agents can audit a repository against its own documentation when no one says which requirement is broken. Each task comes from a real pull request: the diff shows which updated requirements the pre-change code violates, and the updated documentation is written back into the pre-change repository, where nothing marks the target. On 109 tasks with 134 requirements, eight coding agents recover only 35.1–57.5% of the violated requirements. Opus 5 recovers the most, GPT-5.6-Sol has the shortest recorded duration, and GLM-5.3-Flash has the lowest estimated token fees, so the best choice depends on whether coverage, speed, or cost matters most. The trajectories show where proactive investigation falls short. Agents often see a violated promise without checking it: 60.3–92.6% of missed requirements had already appeared in their tool output, and the two agents with the highest recall open the smallest share of the supplied documentation files. Agents also stop verification before the code path that decides the promise. ReqAuditBench measures only whether agents find violations. It does not ask them to fix them. Future work can extend each task so that an agent that finds a violated requirement also changes the code to keep it, and then measure how well these repairs succeed.

AI USE STATEMENT

In this work, we used generative AI tools to implement methods, clean and reformat data and provide feedback on the research methodology and experiments. We have not used generative AI tools to generate synthetic datasets, because every task comes from a real pull request in a maintained repository. Formulating or proving mathematical claims is not applicable to this work. Additionally, we used generative AI tools to create and edit software code, create figures and release artifacts, search for and summarize related work, format references, draft parts of the paper, and edit it for readability.

Within the method, Claude Fable 5 drafts each reference card, and two human reviewers check every card independently (Section 2.3 and Appendix A). Three LLM judges match findings to reference requirements, and Appendix C reports their agreement and the effect of scoring with any single judge. Two LLM auditors, Claude Opus 5.5 and GPT-6-Sol, check sampled findings against the repository snapshots, and one author checked 50 of their judgments (Appendix C.3). Outside the method, LLM coding assistants (Claude Code and Codex) wrote the scripts for candidate mining, task construction, scoring, analysis, and figures under the authors' direction. The evaluated coding agents are the subject of the study rather than tools of the authors.

We have reviewed all AI-assisted work. Verification scripts recompute every result table and result figure from the released records, so no reported result is taken from an assistant's text. The authors linked each claim in the paper to its supporting evidence and checked each cited work against its primary source. We take responsibility for the final content of this work, including text, claims, and artifacts produced with the aid of generative AI.

REFERENCES

Emad Aghajani, Csaba Nagy, Olga Lucero Vega-Márquez, Mario Linares-Vásquez, Laura Moreno, Gabriele Bavota, and Michele Lanza. Software documentation issues unveiled. In *Proceedings of the 41st IEEE/ACM International Conference on Software Engineering (ICSE)*, 2019. doi: 10.1109/ICSE.2019.00122.

Anthropic. Claude API pricing. `https://platform.claude.com/docs/en/about-claude/pricing`, 2026. Accessed September 22, 2026. Model-specific rates cross-checked against the Opus 5 overview.

Syed Ammar Asdaque, Imran Haider, Muhammad Umar Malik, Maryam Abdul Ghafoor, and Abdul Ali Bangash. Novice developers produce larger review overhead for project maintainers while vibe coding. *arXiv preprint arXiv:2602.23905*, 2026. URL `https://arxiv.org/abs/2602.23905`.

Tim Baumgärtner and Iryna Gurevych. SciCoQA: Quality assurance for scientific paper–code alignment. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics*, pp. 38740–38770, 2026. doi: 10.18653/v1/2026.acl-long.1795. URL `https://aclanthology.org/2026.acl-long.1795/`.

Anh T. V. Dau, Jin L. C. Guo, and Nghi D. Q. Bui. DocChecker: Bootstrapping code large language model for detecting and resolving code-comment inconsistencies. In *Proceedings of the 18th Conference of the European Chapter of the Association for Computational Linguistics: System Demonstrations*, 2024. doi: 10.18653/v1/2024.eacl-demo.20.

DeepSeek. Models and pricing. `https://api-docs.deepseek.com/quick_start/pricing/`, 2026. Accessed September 22, 2026. Peak rates used for both DeepSeek models.

Beat Fluri, Michael Würsch, and Harald C. Gall. Do code and comments co-evolve? On the relation between source code and comment changes. In *Proceedings of the 14th Working Conference on Reverse Engineering (WCRE)*, pp. 70–79, 2007. doi: 10.1109/WCRE.2007.21.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *International Conference on Learning Representations (ICLR)*, 2024. URL `https://arxiv.org/abs/2310.06770`.

Tobias Kiecker, Jan Arne Sparka, Martin Reuter, Albert Ziegler, and Lars Grunske. CASCADE: Detecting inconsistencies between code and documentation with automatic test generation. *Proceedings of the ACM on Software Engineering*, 3(FSE):3816–3838, 2026. doi: 10.1145/3808175. URL https://doi.org/10.1145/3808175.

Deepak Kumar. SWE-PRBench: Benchmarking AI code review quality against pull request feedback. *arXiv preprint arXiv:2603.26130*, 2026. URL https://arxiv.org/abs/2603.26130.

Hyeonseok Lee, Gabin An, and Shin Yoo. METAMON: Finding inconsistencies between program documentation and behavior using metamorphic LLM queries. In *LLM4Code*, 2025. URL https://arxiv.org/abs/2502.02794.

Haobin Li, Ping Deng, Weizhong Qian, Liang Jiang, Zhenyu Huang, Mouxing Yang, and Xi Peng. Active-SWE: Benchmarking coding agents for proactive bug fixing without issue reports. *arXiv preprint arXiv:2608.04682*, 2026. URL https://arxiv.org/abs/2608.04682.

Steven Liu, Jane Luo, Xin Zhang, Aofan Liu, Hao Liu, Jie Wu, Ziyang Huang, Yangyu Huang, Yu Kang, and Scarlett Li. TestExplora: Benchmarking LLMs for proactive bug discovery via repository-level test generation. *arXiv preprint arXiv:2602.10471*, 2026. URL https://arxiv.org/abs/2602.10471.

Qinyu Luo, Yining Ye, Shihao Liang, Zhong Zhang, Yujia Qin, Yaxi Lu, Yesai Wu, Xin Cong, Yankai Lin, Yingli Zhang, Xiaoyin Che, Zhiyuan Liu, and Maosong Sun. RepoAgent: An LLM-powered open-source framework for repository-level code documentation generation. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pp. 436–464, 2024. doi: 10.18653/v1/2024.emnlp-demo.46.

Martin Monperrus, Michael Eichberg, Elif Tekes, and Mira Mezini. What should developers be aware of? An empirical study on the directives of API documentation. *Empirical Software Engineering*, 17(6):703–737, 2012. doi: 10.1007/s10664-011-9186-4.

Moonshot AI. Chat model pricing. https://platform.kimi.ai/docs/pricing/chat, 2026. Accessed September 22, 2026. Includes cache-read and retention-dependent cache-write rates.

OpenAI. API pricing and prompt caching. https://developers.openai.com/api/docs/pricing, 2026. Accessed September 22, 2026. GPT-5.6-Sol rates and context tiers cross-checked against the model and prompt-caching documentation.

Sheena Panthaplackel, Junyi Jessy Li, Milos Gligoric, and Raymond J. Mooney. Deep just-in-time inconsistency detection between comments and source code. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2021. URL https://arxiv.org/abs/2010.01625.

Shiva Radmanesh, Aaron Imani, Iftekhar Ahmed, and Mohammad Moshirpour. Investigating the impact of code comment inconsistency on bug introducing. arXiv:2409.10781, 2024.

Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K. Vijay-Shanker. Towards automatically generating summary comments for Java methods. In *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 43–52, 2010. doi: 10.1145/1858996.1859006.

Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. /*iComment: Bugs or bad comments?*/. In *Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pp. 145–158, 2007. doi: 10.1145/1294261.1294276.

Wen Siang Tan, Markus Wagner, and Christoph Treude. Detecting outdated code element references in software repository documentation. *arXiv preprint arXiv:2212.01479*, 2022. URL https://arxiv.org/abs/2212.01479.

Wen Siang Tan, Markus Wagner, and Christoph Treude. Wait, wasn't that code here before? detecting outdated software documentation. *arXiv preprint arXiv:2307.04291*, 2023. URL https://arxiv.org/abs/2307.04291.

Noshin Ulfat, Ahsanul Ameen Sabit, and Soneya Binta Hossain. Measuring LLM trust allocation across conflicting software artifacts. *arXiv preprint arXiv:2604.03447*, 2026. URL https://arxiv.org/abs/2604.03447.

Fengcai Wen, Csaba Nagy, Gabriele Bavota, and Michele Lanza. A large-scale empirical study on code-comment inconsistencies. In *Proceedings of the 27th IEEE/ACM International Conference on Program Comprehension (ICPC)*, 2019. doi: 10.1109/ICPC.2019.00019.

Kai Yu, Zhenhao Zhou, Junhao Zeng, Ying Wang, Xueying Du, Zhiqiang Yuan, Junwei Liu, Ziyu Zhou, Yujia Wang, Chong Wang, and Xin Peng. Does pass rate tell the whole story? evaluating design constraint compliance in LLM-based issue resolution. *arXiv preprint arXiv:2604.05955*, 2026. URL https://arxiv.org/abs/2604.05955.

Z.ai. API pricing. https://docs.z.ai/guides/overview/pricing, 2026. Accessed September 22, 2026.

Yuntong Zhang, Zhiyuan Pan, Imam Nur Bani Yusuf, Haifeng Ruan, Ridwan Shariffdeen, and Abhik Roychoudhury. Code review agent benchmark. *arXiv preprint arXiv:2603.23448*, 2026. URL https://arxiv.org/abs/2603.23448.

Renyi Zhong, Yintong Huo, Wenwei Gu, Jinxi Kuang, Zhihan Jiang, Guangba Yu, Yichen Li, David Lo, and Michael R. Lyu. CCISolver: End-to-end detection and repair of method-level code-comment inconsistency. *arXiv preprint arXiv:2506.20558*, 2025. URL https://arxiv.org/abs/2506.20558.

Qixing Zhou, Jiacheng Zhang, Haiyang Wang, Rui Hao, Jiahe Wang, Minghao Han, Yuxue Yang, Shuzhe Wu, Feiyang Pan, Lue Fan, Dandan Tu, and Zhaoxiang Zhang. FeatureBench: Benchmarking agentic coding for complex feature development. *arXiv preprint arXiv:2602.10975*, 2026. URL https://arxiv.org/abs/2602.10975.

Haichao Zhu, Qian Zhang, Jiyuan Wang, Zhaorui Yang, and Yuxin Qiu. Needle in the repo: A benchmark for maintainability in AI-generated repository edits. *arXiv preprint arXiv:2603.27745*, 2026. URL https://arxiv.org/abs/2603.27745.

A Construction and Task Evidence

A.1 Sources and Inclusion Criteria

The selected repositories are scikit-learn (22 tasks), FastAPI (17), pytest (17), Transformers (15), xarray (15), vLLM (13), MLflow (4), slime (4), and pandas (2). We surveyed 46 Python repositories. Every selected repository has a redistributable license, a non-fork codebase written mainly in Python, complete and clonable history, pull-request metadata hosted on GitHub, user-facing documentation in the form of a documentation site or API pages rendered from docstrings, and a test suite whose CI passes on the default branch. We measure maintenance activity by merged pull requests, because in a pilot on MLflow only about 3% of merged pull requests yielded a task. A repository must have at least 1,000 merged pull requests in the preceding 12 months or, under a 24-month window, at least 600 in the preceding 12 months and at least 150 in the preceding 6 months. Six repositories pass the 12-month threshold, although slime is flagged for declining activity, and FastAPI and pytest pass under the 24-month window. We admitted one exception: xarray, with 360 merged pull requests in 12 months but a high rate of documentation–code co-changes. Counts were retrieved from GitHub in September 2026.

Candidate retrieval follows first-parent history and removes maintenance pull requests using repository-specific label lists. A syntax-tree comparison that strips only standalone docstrings distinguishes implementation changes from wording-only edits. Generated files are excluded, and documentation hunks are paired with implementation hunks through shared code-style identifiers. Eligible documentation includes pages and publicly rendered docstrings or attribute documentation. Changelogs, ordinary comments, and private-object documentation are excluded. The documented public object must already exist in the pre-change code. Target requirements come from added or semantically changed documentation, and each violation is established on the pre-change snapshot. Unchanged passages serve only as context or conditions.

A.2 Reference Authoring and Checks

Claude Fable 5 drafts each reference card. Two reviewers, each with more than three years of software development experience, check every card independently, comparing the documented promise against contradictory source behavior. When their decisions differ, a third reviewer with the same experience resolves the disagreement. Of 243 candidates with drafted cards, 109 remain as tasks. This review establishes reference requirements before agent answers are scored, and it is separate from resolving disagreements between answer graders. The protocol proceeds as follows:

1. Fix the requirements from the pull-request description before reading the diff or code, and identify each public object and its applicability conditions.

2. Pair each eligible documentation block with the implementation change. Read the pre-change code, retrieving pre-change files by targeted `git show` when the diff omits a dependency, and trace the relevant call path to an observable result. Reference behaviors are determined mainly by source reading, and running each behavior is not required.

3. Record the promised and pre-change behaviors with verbatim evidence, requiring a contradiction under the documented conditions, beyond shared names or changed tests. Exclude already satisfied behaviors and mark unsettled ones as undecided.

4. Re-judge undecided behaviors using only the full diff, retaining them only when the diff proves the contradiction (21 of 242 such behaviors across 80 tasks). Confirmed and satisfied behaviors were not re-judged in this pass.

5. Group repeated evidence for the same requirement. Human review checks these semantic judgments, and program checks verify structure, quotation identity, and fingerprints.

A.2.1 Human Review Interface and Worked Example

Human construction review uses an evidence page rather than a reference label alone. The interface presents continuous documentation on one side and the corresponding behavior evidence on the other, with object names, conditions, source locations, and version information. Questions sit below the evidence instead of fragmenting it into isolated answers. In the three-question interface, selecting a question highlights the relevant source lines, and additional context can be expanded without changing its original order. Answers start blank, and reviewers may add notes. The three checks concern object identity, fidelity of the expected behavior to the documentation, and whether the actual behavior contradicts it.

The form offers affirmative, negative, and unsure answers. An unsure answer takes precedence and sends the case for further review. Otherwise, a negative answer prevents acceptance, and all three affirmative answers support acceptance. The interface can retain responses locally and export them as JSON. These are construction-review decisions, not scores for agent answers. The example below phrases the questions for a card whose evidence comes from source inspection rather than execution.

Human construction review: pytest #14751 (adapted form)

Conditions to check. Register an option through `Parser.addini` with `type=int`: the Python type object, not the string `"int"`. The preceding code is inspected under normal execution with assertions enabled.

Requirement evidence
The method's docstring states:

For the scalar types, the plain Python type may be passed instead of the string tag: `str`, `bool`, `int` and `float` (for example `type=int`).

Source: `src/_pytest/config/argparsing.py`, `Parser.addini` docstring.

Behavior evidence
The preceding method body starts with:

```
assert type in (
    None,
    "string",
    "paths",
    "pathlist",
    "args",
    "linelist",
    "bool",
    "int",
    "float",
)
```

Source: the same method's preceding implementation.
Evidence type: source inspection, not a recorded execution.

Reviewer questions (initially unanswered)

1. Do the documentation and behavior evidence concern the same object under the stated conditions?
2. Does allowing `type=int` faithfully express the documented promise in this case?
3. Does the preceding assertion reject that input before the option can be registered?

Each question: ☐ Yes ☐ No ☐ Unsure (optional evidence notes)

The key distinction is between the Python type object `int` and the string `"int"`: finding the latter in the code does not establish support for the former. The allowed values are strings or `None`, so the type object fails the assertion before registration. This explanation is separate from the initially unanswered form. The task is one of the benchmark tasks, and the example illustrates its scalar-type registration behavior, not every behavior under the requirement. The box is a condensed adaptation of the review form using pytest source evidence, not a screenshot or a saved historical verdict. Its documentation and method body are checked against the snapshot, and the full source and version bindings are released with the paper. Every reference card in the benchmark is checked through this human review before agent answers are scored.

A.3 Documentation Regions and Snapshots

Page regions follow documentation navigation and widen toward neighboring sections when small. Docstring regions start with directly contained Python files in the module's directory and widen toward a parent or sibling when needed. Both aim for at least fifteen documentation files where the repository permits it. These rules bound the documentation evidence, while implementation inspection remains repository-wide.

Write-back includes every changed page and existing-object docstring, together with navigation entries for new pages. Tests, changelogs, new-object docstrings, and ordinary comments are excluded. Implementation code remains byte-identical. Modified Python files must also retain identical syntax after docstrings are stripped, and documentation quotations must occur verbatim in the snapshot.

The existing-object criterion applies to the public object, allowing missing documented keywords. For example, xarray-10293 includes a promised `exclude` option on existing equality methods. Later co-changes or additional repositories can reuse the same collection procedure.

Table 3: Evaluated coding agents.

Model	Agent harness	Version	Effort
Claude Opus 5	Claude Code	2.1.69	high
GPT-5.6-Sol	Codex CLI	0.153.4	high
DeepSeek-V4.1-Flash	Claude Code	2.1.69	high
DeepSeek-V4-Pro	Claude Code	2.1.69	high
GLM-5.3	Claude Code	2.1.69	high
GLM-5.3-Flash	Claude Code	2.1.69	high
Kimi-K3	Claude Code	2.1.69	high
HY4-Preview	Claude Code	2.1.69	high

B Agent Setup and Task Instructions

B.1 Evaluated Agents

API model identifiers are `claude-opus-5`, `gpt-5.6-sol`, `deepseek-v4-flash`, `deepseek-v4-pro`, `glm-5.3`, `glm-5.3-flash`, `kimi-k3`, and `hy4-preview`. Display names follow the model versions stated by the API provider. Claude Code requests adaptive thinking with high effort, and Codex CLI requests high reasoning effort. These labels are provider-specific settings rather than a common computation budget. The comparison therefore concerns complete agents.

B.2 Execution Interface

The container provides Python 3.12 and permits shell commands and local scripts. Agents can write scripts under `/work` and import repository code with `PYTHONPATH=/snapshot`. The snapshot is read-only, project dependencies are absent, and package installation is disallowed. Checks requiring additional packages rely on source inspection. The harness sets a two-hour budget, followed by a ten-minute grace period to save the findings file. The task prompt omits this budget. A run is repeated only when it yields no gradable result, for example after a model-service, quota, or runner failure. A run whose findings are graded is never repeated, so each agent has at most one scored run per task, and no score is chosen from several runs. Runs stopped by the operator are not counted as attempts. Each task is run at most three times, and a task without a graded run scores zero. The prompt forbids network access, including raw sockets and IP addresses, and version-history lookup, and the container adds technical restrictions before the agent starts. External name resolution is disabled, so only the model proxy can be reached by name. Python's TLS module and the OpenSSL command are removed, the agent runs as a non-root user without `sudo`, and Claude Code's web fetch and web search tools are denied. GPT-5.6-Sol runs in the same container with Codex web search disabled. Before each run, a check attempts to reach GitHub and PyPI, resolve an external name, and download a package with pip, and the run is aborted if any attempt succeeds. Every recorded check passed. Snapshots are exported without version-control metadata. The container platform cannot block direct TCP connections to IP addresses, so that channel is closed only by the prompt.

B.3 Task Prompt and Submission

The page and docstring templates share the same audit task, five-finding cap, and output fields. The page template restricts documentary evidence to listed documentation files, whereas the docstring template restricts it to docstrings within listed Python files, excluding ordinary comments and `#:` comments. Implementation access is unrestricted within the snapshot.

Figures 5 and 7 give the complete agent task templates for documentation-page and docstring regions. Scoring-judge prompts appear in Appendix C.4. Braced fields are replaced with the documentation-region title, file count, ordered file list, fixed cap $k = 5$, and JSON example. Full original templates and the output example are released with the paper as `evidence/appendix-task-instructions-page`.

`txt`, `evidence/appendix-task-instructions-docstring.txt`, and `evidence/appendix-findings-schema.json`. Their hashes are checked against the fixed methodology and batch configuration. The hash checks verify the released inputs. For every scored run, the recorded prompt matches one of these templates with only the braced fields filled, and the first model request carries that prompt verbatim.

Agents update `/work/findings.json` as each finding is confirmed, using the format in Figure 9. This example fills the {`schema`} field of both templates, and the harness fills `item_id` and `scope_tier`. Scoring uses the submitted findings file and accepts static-reading evidence as well as executed reproductions.

Agent task template: documentation-page region (part 1 of 2)

```
Task type: requirement-level (the documentation describes a
    requirement --- has the code fully delivered it?)

Scope of this task --- {scope_title}
The following {n_files} documentation files form the scope (listed
    in the order of the documentation's own table of contents):
{scope_files}

Among the requirements described in these files, at least one has
    been confirmed as NOT fully delivered by the code. Report every
    one you find (at most {k}, ranked from your highest to your
    lowest confidence). For each one, say which requirement it is
    and which promise the code fails to keep, list every place
    within the scope that states or relies on that requirement
    (file plus verbatim quote), and attach reproducible evidence.
    Look for counterexamples before you conclude.

Scope rule --- read this carefully. ONLY the documentation files
    listed above count for this task. Documentation anywhere else
    in the repository (other pages of the same documentation set,
    other documentation directories, other languages, changelogs,
    README files, docstrings in the Python source) is OUT OF SCOPE:
    you may read it for background, but a finding whose
    documentation location is not one of the listed files is
    invalid and will not be scored, and every entry in its
    `doc_locations` must point to one of the listed files. The code
    that should deliver a requirement may live anywhere in the
    repository --- locating it is part of your job and is not
    restricted.

How to work
- The repository root is /snapshot and it is read-only. There is no
    network and no version history. Do not try to reach the network
    in any way (no raw sockets, no IP addresses, no external
    services), and do not try to identify or look up the original
    change from any source outside /snapshot. Any such attempt
    voids this attempt.
- /work is writable. Put your scripts and drafts there.
- You may read any file and run any command. The container has
    Python 3.12, but this project's dependencies are not installed
    and you cannot install packages. To run code, write a script
    under /work and import with PYTHONPATH=/snapshot; where a
    dependency is missing, judge by reading the code.
- You only need to find the problems, not fix them. Do not try to
    modify anything under /snapshot.
- Do not stop after your first finding: keep checking until you
    believe nothing more is reportable, or you reach the cap of
    {k}. Anything not written to the file does not count.
- **Deliver incrementally**: as soon as you confirm a finding,
    write it into /work/findings.json (overwrite the whole file and
    keep it well-formed), then keep looking. Do not wait until the
    end.
```

Figure 5: Complete agent task template for documentation-page regions (part 1 of 2). Braced fields are filled per task.

Agent task template: documentation-page region (part 2 of 2)

```
Deliverable (mandatory --- without it this attempt is invalid)
- Write the result as JSON to /work/findings.json (update it after
    every new finding) in the format below. One finding = one unmet
    requirement; do not split the same requirement into several
    findings just because the documentation repeats it. `item_id`
    and `scope_tier` are filled in by the evaluator; copy the
    example values as they are.

{schema}

Field notes
- doc_locations: EVERY place within the scope that states or relies
    on this requirement --- one entry per place, each with the file
    path (which must be one of the listed files) and a verbatim
    quote that must be findable by exact search under /snapshot.
    Put the clearest statement first. Two places count as different
    when they are different sentences, examples or sections, even
    in the same file; one requirement found in one place only is
    still a valid finding, but an incomplete list is scored as
    incomplete. Places outside the scope must not be listed.
- claimed_behavior: what the documentation says the library should
    do.
- actual_behavior: REQUIRED. One sentence stating what the code
    actually does, or fails to do, that leaves the requirement
    unmet. A finding without this sentence is not scored.
- code_path / code_symbol: where you believe this promise should
    have been delivered, and the public symbol name. If the
    capability does not exist in the code at all, name the
    function, class or module that should have provided it.
- evidence_kind: static_reading (you judged by reading the code) or
    executed_script (you actually ran a script).
- reproduction: a runnable command or script; null if you have none.
- n_reported must equal the length of the findings array. gave_up
    says whether you gave up. budget_used may stay 0.
- Write your analysis notes to /work/notes.md; they are not scored.
```

Figure 6: Complete agent task template for documentation-page regions (part 2 of 2, continued from Figure 5).

Agent task template: docstring region (part 1 of 2)

```
Task type: requirement-level (the documentation describes a
    requirement --- has the code fully delivered it?)

Scope of this task --- {scope_title}
The following {n_files} Python files form the scope (listed by
    path). "Documentation" for this task means the docstrings
    inside these files (the string literals that open a module,
    class or function); ordinary comments are not documentation:
{scope_files}

Among the requirements described in these files, at least one has
    been confirmed as NOT fully delivered by the code. Report every
    one you find (at most {k}, ranked from your highest to your
    lowest confidence). For each one, say which requirement it is
    and which promise the code fails to keep, list every place
    within the scope that states or relies on that requirement
    (file plus verbatim quote), and attach reproducible evidence.
    Look for counterexamples before you conclude.

Scope rule --- read this carefully. ONLY the docstrings inside the
    files listed above count for this task. Documentation anywhere
    else in the repository (documentation pages, other modules'
    docstrings, other languages, changelogs, README files) and
    comments of any kind (including `#:` attribute comments) are
    OUT OF SCOPE: you may read it for background, but a finding
    whose documentation location is not one of the listed files is
    invalid and will not be scored, and every entry in its
    `doc_locations` must point to one of the listed files. The code
    that should deliver a requirement may live anywhere in the
    repository --- locating it is part of your job and is not
    restricted.

How to work
- The repository root is /snapshot and it is read-only. There is no
    network and no version history. Do not try to reach the network
    in any way (no raw sockets, no IP addresses, no external
    services), and do not try to identify or look up the original
    change from any source outside /snapshot. Any such attempt
    voids this attempt.
- /work is writable. Put your scripts and drafts there.
- You may read any file and run any command. The container has
    Python 3.12, but this project's dependencies are not installed
    and you cannot install packages. To run code, write a script
    under /work and import with PYTHONPATH=/snapshot; where a
    dependency is missing, judge by reading the code.
- You only need to find the problems, not fix them. Do not try to
    modify anything under /snapshot.
- Do not stop after your first finding: keep checking until you
    believe nothing more is reportable, or you reach the cap of
    {k}. Anything not written to the file does not count.
- **Deliver incrementally**: as soon as you confirm a finding,
    write it into /work/findings.json (overwrite the whole file and
    keep it well-formed), then keep looking. Do not wait until the
    end.
```

Figure 7: Complete agent task template for docstring regions (part 1 of 2). Braced fields are filled per task.

Agent task template: docstring region (part 2 of 2)

```
Deliverable (mandatory --- without it this attempt is invalid)
- Write the result as JSON to /work/findings.json (update it after
    every new finding) in the format below. One finding = one unmet
    requirement; do not split the same requirement into several
    findings just because the documentation repeats it. `item_id`
    and `scope_tier` are filled in by the evaluator; copy the
    example values as they are.

{schema}

Field notes
- doc_locations: EVERY place within the scope that states or relies
    on this requirement --- one entry per place, each with the file
    path (which must be one of the listed files) and a verbatim
    quote that must be findable by exact search under /snapshot.
    Put the clearest statement first. Two places count as different
    when they are different sentences, examples or sections, even
    in the same file; one requirement found in one place only is
    still a valid finding, but an incomplete list is scored as
    incomplete. Places outside the scope must not be listed.
- claimed_behavior: what the documentation says the library should
    do.
- actual_behavior: REQUIRED. One sentence stating what the code
    actually does, or fails to do, that leaves the requirement
    unmet. A finding without this sentence is not scored.
- code_path / code_symbol: where you believe this promise should
    have been delivered, and the public symbol name. If the
    capability does not exist in the code at all, name the
    function, class or module that should have provided it.
- evidence_kind: static_reading (you judged by reading the code) or
    executed_script (you actually ran a script).
- reproduction: a runnable command or script; null if you have none.
- n_reported must equal the length of the findings array. gave_up
    says whether you gave up. budget_used may stay 0.
- Write your analysis notes to /work/notes.md; they are not scored.
```

Figure 8: Complete agent task template for docstring regions (part 2 of 2, continued from Figure 7).

Findings submission format

```
{
  "schema": "pilot-detect-findings-v2",
  "item_id": "filled-by-harness",
  "scope_tier": "filled-by-harness",
  "n_reported": 1,
  "gave_up": false,
  "budget_used": {"tool_calls": 0, "wall_seconds": 0,
    "input_tokens": 0},
  "findings": [
    {
      "requirement_summary": "one sentence: what the requirement
    is",
      "doc_locations": [
        {"doc_path": "docs/...", "doc_quote": "verbatim quote from
    the documentation (clearest statement first)"},
        {"doc_path": "docs/...", "doc_quote": "verbatim quote from
    another place that states or relies on the same requirement"}
      ],
      "code_path": "<package>/path/to/module.py",
      "code_symbol": "function, class or attribute name",
      "claimed_behavior": "the documentation says the library
    should ...",
      "actual_behavior": "the code actually ...",
      "evidence_kind": "static_reading",
      "reproduction": null,
      "confidence": "high"
    }
  ]
}
```

Figure 9: Findings example inserted as {schema} in both agent task templates.

C Scoring and Evidence Boundaries

C.1 Requirement Matching and Behavior Coverage

The main score combines mechanical admissibility checks with adjudication of which reference violations a finding identifies. A finding's quotation must appear verbatim in the cited region file, ignoring whitespace differences, and the finding must describe the actual behavior. A semantic match must concern the same promised behavior, object, and conditions, and a correct function name alone is neither necessary nor sufficient. Requirements are counted once even when multiple findings identify them, and scores use all 109 tasks and 134 requirements. Findings outside the reference card are not scored, and Appendix C.3 estimates their precision by model audit. Because every agent is scored on the same registered requirements by the same procedure, recall still supports comparing agents on these targets.

A credited finding covers the behavior that the judges match and every behavior whose reference quotation its in-region quotations overlap. We report requirements with all behaviors covered and matched behaviors pooled over all 263 behaviors. Every recovered requirement has at least one covered behavior. Full coverage means that the finding identifies every registered behavior. Like recall, it is measured against the reference card.

Judge agreement and sensitivity. The eight agents submit 3,217 findings. 201 fail the mechanical check, and 193 are credited by it because their quotation overlaps the evidence of exactly one violated behavior. The remaining 2,823 findings go to the judges. Among them, a finding whose quotation overlaps the evidence of several violated behaviors, and of no other behavior, is already credited by the mechanical check, and the judges only attribute it to one behavior. Each judge returns a matched behavior, no match, or cannot tell. Cannot tell counts as no match, and no judge returned it. The prompt's reference-doubt flag is recorded but not scored. One judge response raised it, and manual inspection found the reference card correct. The three judges give the same match decision on 2,812 findings, with Fleiss' κ of 0.985 and pairwise Cohen's κ between 0.98 and 0.99. None of the 11 split findings involves a reference card with 20 or more behaviors. In 3 findings, at least two judges report a match but no two select the same behavior. 5 judge responses failed validation: two could not be parsed, and three quoted evidence that does not appear verbatim in the finding. They count as no match in these agreement statistics. In scoring, such a finding matches only if the other two judges select the same behavior.

Table 4 recomputes requirement hits under each rule. Scoring with any single judge changes an agent's count by at most 2 requirements and leaves the ranking of Claude Opus 5 first and GPT-5.6-Sol second unchanged. Two evaluated models share a provider with a judge, Claude Opus 5 with Claude Fable 5 and GPT-5.6-Sol with GPT-6-Sol. Excluding that judge and requiring the other two to agree leaves every count unchanged. The released vote record stores all three votes for every judged finding, and the verification scripts recompute every table from it.

C.2 Quotation-Overlap Comparison

Table 5 reports quotation-overlap scoring on the same answers and full benchmark denominators. After checking that a finding contains in-region evidence and an actual-behavior description, it matches the primary quotation to reference spans and groups matches by requirement, without judges. It does not credit an equivalent finding that quotes another supporting passage.

C.3 Findings Outside the Reference Card

Recall counts only registered requirements. To estimate how many submitted findings are real, two LLM auditors, Claude Opus 5.5 and GPT-6-Sol, check a random sample against the repository snapshot. For each agent, we sample 25 uncredited and 5 credited valid findings, 240 in total. Each auditor works independently in a read-only copy of the task snapshot and can run Python. It does not see which agent wrote the finding or what the judges decided. It decides whether the quoted documentation makes the promise, whether the code behaves as the finding states, and whether the two conflict. A conflict counts as real whether the code or the documentation is wrong. We

Table 4: Requirement hits out of 134 under each single judge, the adopted two-of-three majority (bold), and a majority that excludes the judge from the evaluated model's vendor.

Agent	Claude Fable 5	GPT-6-Sol	Qwen3.8-Max	Majority	No same vendor
Claude Opus 5	77	77	77	**77**	77
GPT-5.6-Sol	62	62	62	**62**	62
DeepSeek-V4.1-Flash	60	60	60	**60**	60
DeepSeek-V4-Pro	47	47	47	**47**	47
GLM-5.3	58	58	58	**58**	58
GLM-5.3-Flash	56	56	56	**56**	56
Kimi-K3	49	49	47	**49**	49
HY4-Preview	51	51	52	**52**	52

Table 5: Quotation-overlap scores on the same 109 tasks and 134 requirements.

Agent	RR (%)	THR (%)
Claude Opus 5	71/134 (53.0)	60/109 (55.0)
GPT-5.6-Sol	61/134 (45.5)	52/109 (47.7)
DeepSeek-V4.1-Flash	59/134 (44.0)	50/109 (45.9)
DeepSeek-V4-Pro	47/134 (35.1)	41/109 (37.6)
GLM-5.3	57/134 (42.5)	51/109 (46.8)
GLM-5.3-Flash	54/134 (40.3)	48/109 (44.0)
Kimi-K3	48/134 (35.8)	44/109 (40.4)
HY4-Preview	46/134 (34.3)	42/109 (38.5)

Table 6: Estimated precision of submitted findings. A finding counts as real only if both auditors judge it real. Invalid findings count as not real.

Agent	Submitted	Invalid	Precision
Claude Opus 5	399	12	0.88
GPT-5.6-Sol	506	0	0.79
DeepSeek-V4.1-Flash	394	59	0.74
DeepSeek-V4-Pro	350	34	0.78
GLM-5.3	421	24	0.78
GLM-5.3-Flash	429	15	0.73
Kimi-K3	404	37	0.85
HY4-Preview	314	20	0.78

estimate each agent's precision as the share of all its submitted findings judged real. The credited and uncredited sample rates are weighted by their counts, and invalid findings count as not real.

Table 6 reports the results. When a finding counts as real only if both auditors judge it real, estimated precision ranges from 0.73 to 0.88. No agent's recall comes from indiscriminate reporting. Both auditors judge all 40 sampled credited findings real. Neither auditor maps any of the 200 sampled uncredited findings to a reference behavior. We do not report F1, because the reference card does not list every inconsistency in a region.

Recall is therefore defined by what the maintainers acted on. Every registered violation is a behavior that the maintainers judged worth a merged pull request, whether by fixing the code, adding the documented capability, or completing a deprecation. A finding outside the reference card may still be a real inconsistency, but no maintainer change confirms that it matters, so recall does not credit it. Recall thus measures whether an agent finds the inconsistencies that the project itself chose to resolve.

The auditors agree on 90.4% of real-or-not decisions. Cohen's κ is 0.49, low because most findings are real. GPT-6-Sol judges more findings real than Opus 5.5. Each auditor shares a provider with one evaluated agent, and GPT-6-Sol is also a judge. For each auditor, its rate for the agent from its own provider falls within its range for the other agents. In one task, both auditors read third-party library source outside the snapshot. Excluding that task changes one estimate by 0.03.

One author labeled 50 findings as real or not. The 50 are 30 random sample findings and 26 on which the auditors disagree, with 6 in both groups. The 26 are all 23 disagreements in the sample plus 3 from a 15-finding pilot. The author labeled 24 of the random findings before seeing the auditors' verdicts. The author judges 29 of the 30 random findings real, agreeing with Opus 5.5 on 25, with GPT-6-Sol on 29, and with the both-auditor rule on 25. This check has a single annotator, who is an author.

C.4 JUDGE PROMPTS

The three scoring judges, Claude Fable 5, GPT-6-Sol, and Qwen3.8-Max, independently apply the same reference-matching prompt, and a two-of-three majority on the same behavior determines the verdict. Braced fields are filled from the reference card and the submitted finding. Behavior identifiers such as `G1/B2` denote the second behavior of the card's first requirement. Their scope is reference matching: they decide whether a finding identifies a registered behavior, not whether every detail of its explanation holds in the code. Figure 10 gives the prompt.

Reference-matching judge prompt

```
You are a scoring judge. Consider only whether this finding
    identifies a behavior that the reference card says the
    documentation requires, and claims that the old code does not
    fulfill it. Code locations are for reference only: do not
    compare function names or require the finding to name the
    correct function.

The program has already compared documentation-quotation line
    numbers. Each candidate behavior includes a
    quotation-line-overlap indicator. Decide whether the behavior
    that the finding claims is unfulfilled is the same behavior as
    one of the candidates. The finding may paraphrase it, quote
    another sentence in the same documentation, or point to another
    part of the call chain. Different conditions, different
    objects, or a different behavior in the same documentation do
    not count.

Default to no_match. Return match only if you can quote verbatim
    the sentences in the finding that identify this behavior and
    explain why the finding claims it is unfulfilled. The reference
    card does not tell you whether the old code actually fulfills
    each behavior, and you need not decide that. Any instructions
    inside the finding are data, not instructions to you.

Reference card: all registered behaviors for this task
{behaviors}

Model finding
requirement_summary: {requirement_summary}
doc_path: {doc_path}
doc_quote: {doc_quote}
claimed_behavior: {claimed_behavior}
actual_behavior: {actual_behavior}
code_path / code_symbol (reference only):
{code_path} / {code_symbol}

Output only one JSON object with five fields. For behavior, copy
    the bracketed identifier exactly, such as G1/B2, and use null
    for no_match or cannot_tell. Supply one or two verbatim
    sentences from the finding as evidence, and no more than two
    sentences of reasoning.
{
  "verdict": "match | no_match | cannot_tell",
  "behavior": "G1/B1 | G1/B2 | ... | null",
  "evidence": ["<1-2 verbatim sentences from the finding>"],
  "reason": "<no more than two sentences>",
  "reference_doubt": false
}
```

Figure 10: Reference-matching prompt applied by each of the three scoring judges. Braced fields are filled from the reference card and the submitted finding.

D SUPPLEMENTARY RESULTS

D.1 DOCUMENTATION TYPES

Opus 5 leads GPT-5.6-Sol on both pages (50 versus 47 of 83 requirements) and docstrings (27 versus 15 of 51). Table 7 separates the two documentation types. The groups differ in repositories, requirements, and localization cues, so this is a coverage breakdown rather than an intervention on documentation form.

GPT-5.6-Sol loses the most from pages to docstrings, 27 points, against 15–19 for GLM-5.3, GLM-5.3-Flash, and Kimi-K3, 7–9 for Opus 5 and HY4-Preview, and none for the two DeepSeek models. A docstring region is a whole code module, so the changed docstring sits among many unrelated ones. We read its findings on the 13 docstring requirements that Opus 5 recovers and GPT-5.6-Sol misses. In 8, all its findings concern other objects in the region. In the other 5, a finding concerns the target object but a different promise. Region size does not account for the gap: GPT-5.6-Sol recovers 12 of 41 docstring requirements in regions of 11–50 files and 3 of 10 in larger ones.

D.2 TASK TYPES

Table 8 breaks recall down by the task types of Section 2.4. Labels are assigned per task, and a task's requirements share them. Claude Fable 5 labeled all tasks in two requests from each pull request's title and labels and from each requirement's object, condition, and promised and pre-change behavior, and the authors then checked every label. Opus 5 leads GPT-5.6-Sol on new-feature requirements, with 18 requirements recovered only by Opus 5 against 6 only by GPT-5.6-Sol. For the other three intents, neither agent has more than three requirements that only it recovers.

D.3 MODEL-FREE LOOKUP BASELINE

This baseline calls no model and tests whether the registered requirements can be found by looking documented names up in the code. It receives each task's documentation region and snapshot and reports documented items that the code does not provide. The first rule reports identifier-like mentions that occur nowhere in the snapshot code: backquoted names, documentation directives, command-line flags, environment variables, file names, and `name=value` settings. The second reports documented calls whose keyword argument no definition of the called function accepts. The third applies to docstring regions. It reports documented parameters missing from the signature, documented literal defaults that differ from the signature default, and documented exceptions that the function body never mentions. Signature rules rank first, and the first five candidates form the answer, which is scored like the agents' answers under the same five-finding cap and three-judge rule. The baseline never chooses which requirement matters and never follows implementation logic.

It recovers 4 of 134 requirements (3.0%) on 3 of 109 tasks (2.8%), against 35.1% to 57.5% for the agents, and the judges add no requirement to those the mechanical check already credits. All four come from quotation overlap. Two, on one xarray task, come from one finding that reports a documented parameter missing from the signature. The other two come from findings that report a different missing name but quote a passage overlapping the reference evidence.

D.4 COVERAGE WITHIN REQUIREMENTS

Table 9 asks how completely findings cover a recovered requirement, beyond the at-least-one-behavior criterion used in the main score. It reports requirements with all behaviors covered and the number of covered behaviors out of 263. For every agent, fewer requirements have all behaviors covered than have at least one covered. Findings thus leave parts of some recovered requirements unreported. These counts measure coverage of the reference card (Appendix C).

Table 7: Requirement recall (RR) by documentation type: 83 page and 51 docstring requirements.

Agent	Page RR (%)	Docstring RR (%)
Claude Opus 5	50/83 (60.2)	27/51 (52.9)
GPT-5.6-Sol	47/83 (56.6)	15/51 (29.4)
DeepSeek-V4.1-Flash	37/83 (44.6)	23/51 (45.1)
DeepSeek-V4-Pro	27/83 (32.5)	20/51 (39.2)
GLM-5.3	42/83 (50.6)	16/51 (31.4)
GLM-5.3-Flash	40/83 (48.2)	16/51 (31.4)
Kimi-K3	35/83 (42.2)	14/51 (27.5)
HY4-Preview	35/83 (42.2)	17/51 (33.3)

Table 8: Requirement recall (%) by task type. Columns: Opus = Claude Opus 5, Sol = GPT-5.6-Sol, DS-F/DS-P = DeepSeek-V4.1-Flash/V4-Pro, GLM/GLM-F = GLM-5.3/GLM-5.3-Flash, Kimi = Kimi-K3, HY4 = HY4-Preview.

Type	Tasks	Req.	Opus	Sol	DS-F	DS-P	GLM	GLM-F	Kimi	HY4
Pull-request intent										
New feature	48	62	64.5	45.2	46.8	43.5	50.0	48.4	41.9	41.9
Bug fix	28	31	48.4	54.8	48.4	32.3	29.0	35.5	35.5	35.5
Deprecation or removal	15	17	64.7	52.9	41.2	35.3	52.9	35.3	41.2	41.2
Other behavior change	18	24	45.8	33.3	37.5	16.7	37.5	37.5	20.8	33.3
Violation form										
Capability missing	51	67	58.2	47.8	41.8	37.3	49.3	46.3	40.3	43.3
Behavior deviates	58	67	56.7	44.8	47.8	32.8	37.3	37.3	32.8	34.3

Table 9: Reference-coverage depth: requirements with all behaviors identified, and total behaviors identified.

Agent	Reqs. with full behavior coverage (#)	Behavior coverage (%)
Claude Opus 5	71	153/263 (58.2)
GPT-5.6-Sol	59	130/263 (49.4)
DeepSeek-V4.1-Flash	54	120/263 (45.6)
DeepSeek-V4-Pro	39	82/263 (31.2)
GLM-5.3	54	115/263 (43.7)
GLM-5.3-Flash	48	104/263 (39.5)
Kimi-K3	42	108/263 (41.1)
HY4-Preview	49	91/263 (34.6)

E Evidence for Proactive Investigation

E.1 Case Selection and Recall

The Array API group contains nine requirements from nine distinct scikit-learn pull requests. Within the three-run limit, the eight agents have 71 completed runs: HY4-Preview has none on sklearn-34785 because its runs failed, so that requirement counts as a miss and every recall in Table 10 is out of nine. Every run displays the group's shared documentation statement, so Table 10 reports only recall. This group supplies a shared documentation promise for examining implementation checks, and conclusions are limited to this group.

The paired study takes all five Opus 5-recovered/GPT-5.6-Sol-missed requirements in this group, covering ten original trajectories: PRs 32582, 32586, 32597, 32613, and 32985. We manually reviewed these paired trajectories and confirmed Opus 5's target-specific implementation or test checks in each pair. This outcome-selected comparison characterizes the observed investigation pattern in these five pairs, rather than estimating how often it occurs across all benchmark tasks. We assess implementation inspection from the source shown in the trajectories, including checks based on source lines rather than function names, such as the GLM-5.3-Flash success below.

The excerpts include the Opus 5–GPT-5.6-Sol pair for sklearn-32597, complementary success and failure cases in sklearn-32613, sklearn-31671, and sklearn-34785. They provide selected examples rather than exhaustive semantic annotation of the 71 runs.

E.2 Source Excerpts

A shared promise, different reported targets. These selected excerpts expand Section 4. They describe visible actions, not population-wide or causal differences. In scikit-learn PR 32597, both tool outputs include "Estimators and other tools in scikit-learn that support Array API compatible inputs." and list `sklearn.metrics.pairwise.manhattan_distances`. The page requires `SCIPY_ARRAY_API` to be set to `1` before imports and explicit `array_api_dispatch` configuration. Under this setup, the documented convention is to preserve the input library and device. Opus 5 reads the dense branch, `return distance.cdist(X, Y, "cityblock")`, the `"manhattan": manhattan_distances,` mapping, and the single-worker `return func(X, Y, **kwds)` path. Its final report states "manhattan_distances contains no array API handling at all". GPT-5.6-Sol's tool outputs include shared namespace utilities and `mean_squared_error`, and its five findings include `Ridge.fit` and `mean_squared_error`, not the Manhattan-distance promise. Its tool outputs show `manhattan_distances` only in the documentation list; its searches over `pairwise.py` return namespace call sites of other functions, and none displays the body of `manhattan_distances` or its dispatch mapping. Hidden reasoning and truncated returns are not observable.

A guessed helper versus a checked operation. In PR 31671, DeepSeek-V4.1-Flash reads the documented `d2_absolute_error_score` and `d2_pinball_score` candidates, then writes "d2_pinball_score has a `_d2_pinball_score` internal and uses xp presumably." A later scripted keyword search again flags both functions, and the agent calls the heuristic "noisy" and concludes "I'll stop the metric sweep." The search scans fixed-length source windows for `get_namespace`, and its `False` outputs are search results, not failed metric executions. Neither target appears in its five final findings. These excerpts preserve the return to the candidates rather than treating the first change of direction as the end. By contrast, GLM-5.3-Flash reads `laplacian_kernel` by line range in PR 32613. Its tool outputs show `np.exp(K, K)` versus `_modify_in_place_if_numpy(xp, xp.exp, K, out=K)` nearby. Its final report identifies `laplacian_kernel`, states that it "never acquires the namespace", and names the NumPy operation. These are static source checks, not executions of the target functions.

An unfinished check by Opus 5. In PR 34785, Opus 5 reads the documented `contingency_matrix` implementation, including `np.unique(labels_true, return_inverse=True)` and `sp.coo_array`. Its note says "This feels like it needs more digging". It then reads Array API upcoming-change notes and later lists `contingency_matrix` among "More candidates to check:", but its four final findings concern estimator support tags

Table 10: Requirement recall within the nine-task Array API group.

Agent	Recovered requirements
Claude Opus 5	8/9
GPT-5.6-Sol	3/9
DeepSeek-V4.1-Flash	0/9
DeepSeek-V4-Pro	4/9
GLM-5.3	4/9
GLM-5.3-Flash	6/9
Kimi-K3	2/9
HY4-Preview	5/9

Table 11: Missed requirements with a reference quotation displayed in a tool output. Percentages use the missed requirements in each row.

Agent	Missed requirements	Target quote shown
Claude Opus 5	47	36 (76.6%)
GPT-5.6-Sol	68	41 (60.3%)
DeepSeek-V4.1-Flash	68	63 (92.6%)
DeepSeek-V4-Pro	81	70 (86.4%)
GLM-5.3	76	69 (90.8%)
GLM-5.3-Flash	78	66 (84.6%)
Kimi-K3	82	72 (87.8%)
HY4-Preview	56	49 (87.5%)

instead. The source record therefore extends beyond the tentative note, but the candidate still does not reach the final report. This omission is not a failure to encounter the entry or function body, nor evidence of a completed behavioral check.

E.3 Target-Quotation Exposure Protocol

The exposure analysis uses the documentation region from each task and the tool outputs actually included in recorded model requests, including visible subagent contexts. Only displayed source text counts as documentary exposure, and read commands, file names, assistant paraphrases, and inferred reading labels are excluded. Hashes of each snapshot archive and its extracted files bind the source text to the snapshot.

For each requirement, we collect quotations from its verified inconsistent reference behaviors. A quotation is eligible when it contains at least 40 characters after whitespace normalization and can be attributed to only one file within the supplied documentation region. Matching removes tool-added line-number prefixes and normalizes whitespace within each tool output. A requirement is exposed when at least one eligible quotation appears completely within one return. Separate returns remain separate, and a match may cover only part of a multi-condition promise.

For each agent, the numerator counts missed requirements with a confirmed match. The denominator includes every missed requirement in a completed run, including those without an eligible quotation. A task with no completed run within the three-run limit (Appendix B.2) scores zero in the main results but leaves no trajectory to analyze, so its requirements enter neither count. Per-match records preserve the reference quotation, source location, tool-output position, and hashes, and the counts are recomputed mechanically without a model-based reading or failure-stage label.

For paired Opus 5–GPT-5.6-Sol trajectories, 22 of the 26 requirements recovered only by Opus 5 have a reference quotation in GPT-5.6-Sol's tool outputs. This overlap motivates examining how the agents used the displayed text to investigate the promises.

E.4 Region-Coverage Protocol

Figure 4 is computed from the 833 completed runs within the three-run limit. A turn is one model request by the main agent that issues at least one tool call. A file counts as opened when the main agent reads it with the Claude Code `Read` tool or with a shell read command (`cat`, `sed`, `head`, `tail`, `nl`, `less`, `more`, `bat`, or `awk`). The path is taken from the command's last file argument, and paths containing a shell variable or wildcard are not counted. An opened path belongs to the region when it and a region entry match as path suffixes. Search commands such as `grep`, `rg`, and `find` do not open a file even when their output shows its lines, and commands after a pipe are ignored. All rules act on tool names and command text, and no model reads the trajectories.

Each curve averages over the 76 tasks with a recorded run from every agent, so all agents face the same regions. A run that has ended keeps its final coverage at later turns, so late turns are not limited to long runs. The plot stops at turn 120, while final coverage in Section 4.1 uses each full run. The rank correlation with recall uses the requirement recall of Table 2 and has eight points.

Only reads by the main agent count. In Claude Code traces, subagent requests carry a smaller tool list than the main agent's and are excluded. Codex CLI traces do not record subagent threads. Runs that start subagents therefore understate reading. As a check, we repeat the turn-10 measurement over all tasks on runs that start no subagent (21–85 runs per agent). Opus 5 and GPT-5.6-Sol reach 20–23% versus 44–60% for the other six agents, and the rank correlation with recall remains $\rho = -0.76$.

This measure differs from the exposure analysis above. Exposure asks whether a missed requirement's reference quotation appeared in any tool output, including search results and subagent contexts. Region coverage asks how many region files the main agent chose to open, whether or not they contain a target.

F Resource and Cost Accounting

F.1 Recorded Usage and Denominators

Table 2 and Figure 3 report arithmetic means over the recorded trajectories of scored runs, including those with no recovered requirement. These resource summaries cover only the recorded trajectories of scored runs, whereas scores use all 109 tasks and 134 requirements.

Input tokens in Table 2 sum ordinary input, cache writes, and cache reads without double counting. Output tokens are reported separately. Each mean divides the corresponding token total by the number of recorded trajectories for that agent. We count model responses, not tool calls. Recorded duration spans the first request to the last response, excluding provisioning. Harnesses, provider accounting, and the tasks represented by resource records differ across agents, so these are observational comparisons rather than controlled efficiency estimates.

F.2 Total Token Fees

Table 12 sums ordinary input, cache writes, cache reads, and output. We partition input usage into disjoint billing buckets, apply the relevant per-million-token rate to each request, and sum before averaging over runs. For Opus 5, the \$28.13 mean total comprises about \$4.11 in output, \$14.18 in cache writes, \$9.84 in cache reads, and \$0.002 in ordinary input. Output-only pricing would therefore substantially understate its token fees. These are public-price estimates, not actual bills or the total budget of all experimental attempts. They exclude machine provisioning, grading, and taxes.

F.3 Price Schedule and Cache Accounting

Table 13 gives the public standard rates retrieved on September 22, 2026, from provider documentation (Anthropic, 2026; OpenAI, 2026; DeepSeek, 2026; Z.ai, 2026; Moonshot AI, 2026). HY4-Preview uses standard rates supplied by the authors on September 24, 2026, for which no public price page is cited. DeepSeek uses a uniform peak-price schedule and excludes time-of-day discounts.

Table 12: Mean token fees per recorded run (USD), using the standard rates in Table 13. Billing buckets are disjoint.

Agent	Input	Write	Read	Output	Total
Claude Opus 5	0.002	14.176	9.841	4.107	28.127
GPT-5.6-Sol	0.648	0.462	2.977	0.658	4.745
DeepSeek-V4.1-Flash	0.365	0.000	0.177	0.253	0.795
DeepSeek-V4-Pro	0.789	0.000	0.556	0.424	1.769
GLM-5.3	1.417	0.000	4.875	0.691	6.983
GLM-5.3-Flash	0.102	0.000	0.370	0.048	0.520
Kimi-K3	1.535	1.251	5.906	2.241	10.932
HY4-Preview	2.093	0.000	1.549	0.732	4.373

Table 13: Standard rates in USD per million tokens. Paired cache-write rates denote five-minute / one-hour retention. Other cache-write tokens use the input rate.

Agent	Input	Cache read	Cache write	Output
Claude Opus 5	5	0.5	6.25 / 10	25
GPT-5.6-Sol	4	0.4	5	20
DeepSeek-V4.1-Flash	0.3	0.006	same as input	1.2
DeepSeek-V4-Pro	1.32	0.044	same as input	3.96
GLM-5.3	1.4	0.26	same as input	4.4
GLM-5.3-Flash	0.15	0.03	same as input	0.5
Kimi-K3	3	0.3	3 / 6	15
HY4-Preview	1	0.05	same as input	3

Opus 5's recorded cache writes all use one-hour retention. Kimi supports five-minute and one-hour writes. Where response usage omits the duration, matching request identifiers recover the five-minute cache setting. The other providers' cache-miss or write tokens use the ordinary input rate unless a distinct write rate is shown in the table. Request-level context checks find no GPT-5.6-Sol request above its 272,000-token price threshold. The released evidence retains rate snapshots, disjoint token totals, and source hashes for the recorded trajectories.

DeepSeek-V4.1-Flash consumes more responses and output tokens per recorded run than Opus 5 while recovering fewer requirements, but this does not show that reducing a given agent's budget would improve its results.