

DOGBENCH: Can Agents Meet Expert Standards for User-Facing Documentation?

Frances Liu*
PROMPTLESS

Manny Silva
PROMPTLESS / DOC DETECTIVE

Paige Calvert
HELM

Ayu Adiati
MAUTIC

Sarah Sanders
POSTHOG

Abstract

We introduce DOGBENCH (Documentation Generation Benchmark), to our knowledge, the first benchmark for generating and maintaining real user-facing software documentation. It asks whether an agent can produce documentation that experienced technical writers would accept in review. The benchmark contains 292 items from open source projects, including Helm, PostHog, and Mautic. Each item gives the agent a pre-change repository and a trigger, such as a code pull request or a reported documentation gap. The agent must first decide whether the documentation needs an update. For items that need one, the agent must produce an acceptable patch in one attempt. For items that do not need updates, the agent must abstain. Task-specific rubrics, validated with project maintainers, score each patch on accuracy, completeness, reader guidance, placement, and repository conventions. The composite score combines patch quality with correct abstention, and a score of 100 means an agent meets every requirement for the task. Scores should not be interpreted as a percentage of an expert’s capability. We evaluated seven agents. The highest-scoring agent reached 47.3 out of 100 on the 117-item held-out split. In a separate audit of 1,267 patches, the most common failure modes were task-completion gaps (45.5%), technical inaccuracies (36.6%), and incomplete conceptual or reference coverage (32.5%). Analysis of the corresponding trajectories identified three key patterns associated with these failures: (1) describing interfaces without examining how readers use them (36.0%), (2) missing decisive evidence and filling the gaps with plausible assumptions (33.1%), and (3) stopping after finding the first plausible documentation surface and leaving other affected pages stale (30.1%).

1 Introduction

User-facing documentation is the main public description of what a software product is, what capabilities it offers, and when and how to use them. For closed-source products in particular, documentation may be the only structured source an AI agent can use to understand and operate the product. People increasingly rely on AI agents to find, choose, and use products. Documentation therefore affects whether an agent uses a product correctly and whether the agent considers the product for the user’s task at all.

Producing this content requires more than translating implementation details into prose. The work demands judgment about where information belongs, what readers are trying to accomplish, and what they already know. Teams increasingly use agents to write documentation, but no one has yet systematically evaluated the user-facing documentation that agents produce.

Prior work evaluates code-facing documentation rather than user-facing documentation (Section 2.1). That work covers function-level docstrings, repository-level code summaries, and internal developer documentation, and the work assumes that a human already decided that the documentation needs an update. No existing documentation benchmark tests whether an agent can tell when to leave the documentation alone.

*frances@promptless.ai

We introduce DOGBENCH, a benchmark of 292 items from open source projects. Each item gives an agent a pre-change repository and a trigger, which is either a pull request or a reported documentation gap. The agent first decides whether the trigger calls for a documentation change. For 205 items, the correct response is a patch. For the other 87, the correct response is to abstain. Task-specific rubrics, validated with project maintainers, score each patch. The rubrics do not reward similarity to the documentation that humans merged. Depending on the task, a correct patch may revise existing guidance, create and register a new page, move content, or remove stale or redundant content. Every item starts from an existing product and its then-current documentation. The benchmark therefore does not cover writing a product’s documentation from scratch or redesigning an information architecture without constraints.

We use a random, stratified 117-item held-out split for primary evaluation. The other 175 items form a public development split. We use the public split and the full 292 items only for robustness analyses.

We evaluated seven agent lanes. The highest-scoring agent reached 47.3 out of 100 on the held-out split. Only 6.1% of submissions contained fabricated content. The more common failure was a plausible patch that left the reader unable to finish the task. Of 1,267 submissions, 45.5% had a task-completion gap. Section 7 traces these failures to how agents investigate. In 36.0% of submissions, agents explained product interfaces without checking how readers use them. In 33.1%, agents stopped before finding decisive evidence. In 30.1%, agents edited the first plausible documentation surface and missed other surfaces the change affected.

2 Related Work

2.1 Documentation generation benchmarks

Existing documentation benchmarks focus on code-facing documentation. CodeSearchNet supplied a corpus of paired functions and documentation for semantic code search, and CodeXGLUE used CodeSearchNet-derived data for code summarization [3, 10]. More recent benchmarks study docstring updates after code changes (CoDocBench) and repository-level internal documentation (CodeWikiBench) [12, 15]. Like DOGBENCH, SWD-Bench builds its tasks from pull requests, but it scores repository-level documentation by how well a model can use that documentation to answer questions about the repository’s functionality [19]. None of these benchmarks asks the model to decide whether documentation needs an update.

2.2 Scoring open-ended edits

SWE-bench, which also draws from open source repositories, is widely used to evaluate code generation [4]. OpenAI has since questioned the validity of its Verified subset because narrow tests can reject correct alternative solutions [9]. That risk is larger for documentation because many different edits can satisfy the same reader need. DOGBENCH therefore scores each patch against requirements drawn from the triggering change and the pre-change repository rather than against similarity to the merged human patch.

3 Benchmark Design

Each item in the benchmark gives the agent a pre-change repository and a trigger. The trigger is either a code pull request or a user-reported documentation gap, often a GitHub issue. This setup mirrors how maintainers work. A maintainer either ships documentation changes with a feature or updates the documentation in response to a community issue. The agent must either return a patch that edits the documentation or abstain from making documentation changes when no user-facing change is needed. Figure 1 summarizes the item-construction and evaluation pipeline.

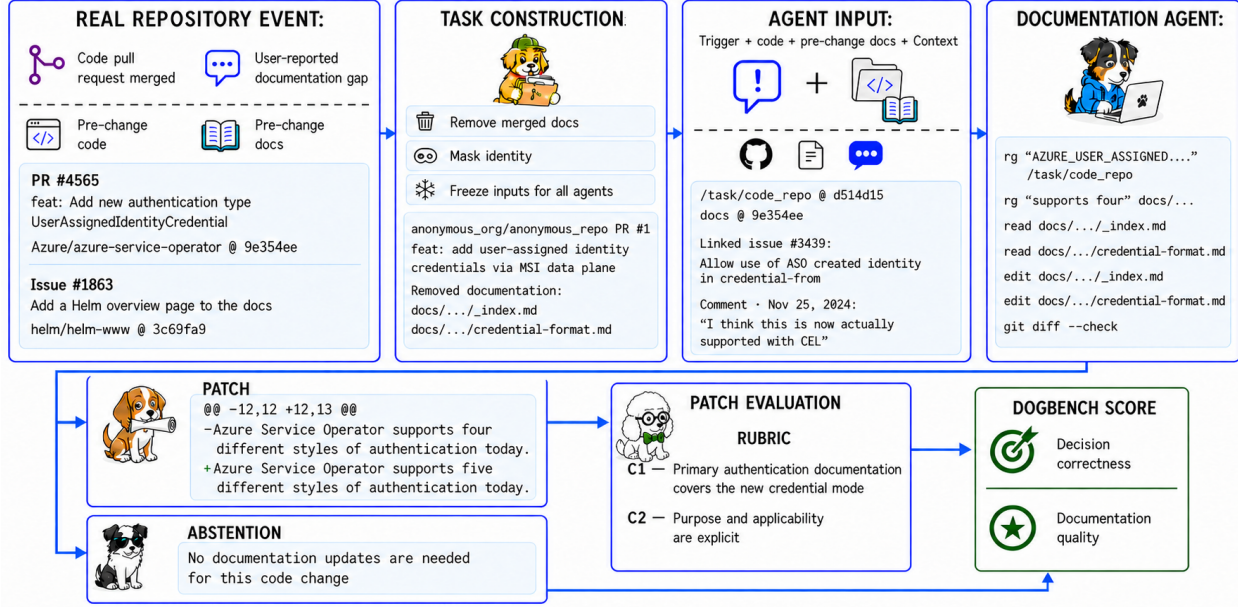


Figure 1: Overview of DOGBENCH. Each item begins with a real trigger event and a frozen, identity-masked snapshot of the pre-change repository. A documentation agent either abstains or emits a patch, and task-specific rubrics score the patch. DOGBENCH reports decision correctness and documentation quality separately. Its composite score combines delivered patch quality with abstention recall.

3.1 Dataset construction

The benchmark contains 292 items: 205 require a documentation change, and 87 require abstention. We built them from three pools. The first holds 90 changes initially sampled as likely abstention cases. The second holds 136 code-triggered documentation updates, and the third holds 66 explicit user-reported documentation gaps. During final adjudication, we reclassified three of the 90 likely abstention cases as requiring documentation, which left 87 abstention items. The final set therefore has 139 code-triggered documentation items, 66 items triggered by reported documentation gaps, and 87 abstention items. Project maintainers reviewed 42 items from repositories such as Helm, Doc Detective, Mautic, and PostHog. Appendix A gives additional selection details and two case studies.

Constructing the trigger

For a pull request that ships code and documentation together, we remove the documentation changes and give the agent only the code change. For a pull request that changes only documentation, we build the trigger from linked issues and discussions. We add sanitized versions of the source pull request’s title and description. Both methods keep the documentation need and the reason for it but hide how the maintainer wrote the documentation.

We selected open source repositories with English-language documentation across diverse ecosystems. We exclude the following:

- reverts, release-only version bumps, and merge or sync pull requests
- pure refactors
- changes where the connection between trigger and the documentation need is unclear
- items that need context unavailable in the public repository or trigger

We also exclude bot-authored pull requests unless a human maintainer reviewed and revised the

change before merging. Appendix A gives the full rules and known limitations.

3.2 Contamination controls

Because the source events are public, contamination can happen if a model has seen the merged documentation during training or if an agent finds it while running. We probe training-time exposure with source-event-date and repository-footprint ablations (Section 6.3) [16]. To prevent execution-time exposure, agents work in fresh Docker containers with identity-masked repositories and no network access except to the model provider. We also inspect agent trajectories for attempts to reach the merged human patch. A conservative overlap detector flags suspicious similarity to the merged documentation for manual review. We found no confirmed case of copying. We reviewed every retained detector alert and judged each one a false positive.

4 Evaluation Protocol

For each item that requires a documentation update, we score the submitted patch against a task-specific rubric. Across the 205 documentation-needed items, the rubrics contain 3,273 criteria, including 798 P0 criteria.

We design each criterion to test one observable review decision. For example, suppose a change lets a Helm values file to contain multiple YAML documents. Separate criteria can then check the documentation for three statements: documents are processed in order, later values take precedence, and nested maps are merged recursively. A broad criterion such as “explains multi-document values well” would not be testable enough.

4.1 Criterion types and priorities

Each criterion use one of three scoring types:

- A **requirement** always applies and receives a binary Pass or Fail verdict.
- A **conditional criterion** applies only when the patch meets its condition. We call a conditional criterion *triggered* when it applies.
- A **deduction-only guardrail** prohibits content such as a fabricated command or an unsafe recovery step. Avoiding the prohibited content earns no credit, and introducing it costs a deduction.

Requirements and triggered conditional criteria receive only Pass or Fail verdicts, with no partial credit. Untriggered conditional criteria are left out of scoring.

Each criterion also has a priority from P0 to P3, which is separate from its scoring type. P0 is reserved for a defect that blocks the patch on its own: failing that criterion alone would require revision under the benchmark’s standard. Typical P0 defects include the following:

- materially misstating product behavior
- omitting information the reader needs to complete the central reader task
- giving an unsafe or destructive instruction
- inventing a public interface
- leaving an essential maintained documentation surface contradictory or unusable

Optional examples, secondary edge cases, stylistic preferences, and exact wording do not qualify as P0 merely because they would improve the patch. A patch is *P0-clean* when no applicable P0 criterion failed. P0-clean diagnoses only critical defects and does not a prediction whether a maintainer would merge the patch unchanged.

4.2 Score construction

Let R contain all requirements and triggered conditional criteria, and let G contain all violated deduction-only guardrails. Each criterion in R contributes one point if it passes, and each violated guardrail deducts one point. For $|R| > 0$, the uncapped patch score is

$$\tilde{q} = 100 \frac{\max(0, \sum_{i \in R} \mathbf{1}[i \text{ passes}] - |G|)}{|R|}.$$

An untriggered conditional criterion counts in neither the numerator nor the denominator. A guardrail that the patch does not violate is also left out, so a patch earns no points merely for avoiding an optional risk. If R is empty, the score is zero.

Let $B = 1$ when a P0 requirement or triggered conditional criterion fails, or when a P0 guardrail is violated. Otherwise, let $B = 0$. The reported patch score is

$$q = \begin{cases} \min(\tilde{q}, 60), & B = 1, \\ \tilde{q}, & B = 0. \end{cases}$$

On a documentation-needed item, an empty patch or an abstention also scores zero. We set the 60-point ceiling as an evaluation policy. We did not estimate it from maintainer editing time or acceptance decisions. The ceiling prevents success on many secondary criteria from averaging away a critical defect.

4.3 Rubric construction

Research and synthesis

We build each rubric through repository research and an LLM-council process. The merged human patch is not included among the candidate patches supplied to the rubric agents. A rubric research agent inspects the triggering change and the repository to identify what users need to know and the evidence that supports each requirement. The research agent has internet access and may encounter the merged human patch, but every criterion must have independent supporting evidence; the human patch alone cannot justify a criterion. A synthesis stage turns these findings into criteria with explicit passing and failing conditions. Mei et al. [13] also explore this research-to-criteria approach.

Differential review

The differential stage compares anonymous candidate patches side by side to find editorial choices that the draft rubric does not yet cover. This stage follows the observation that inspecting model outputs can help refine evaluation criteria [17]. For each uncovered difference, the rubric agent checks the research findings and gathers more evidence where needed. It then decides whether the difference matters to the reader. A difference between candidates only raises a question and does not establish what is correct. Trivial or neutral differences do not become criteria. The rubric agent also records supported documentation needs that no candidate meets. These findings are then used to revise the draft rubric.

Audit and debate

A second model, from a different model family, audits the revised criteria and their priorities. When the rubric author and the auditor disagree, they revisit the evidence and exchange arguments. Together they revise or remove any criterion that cannot be justified. The debate ends when the auditor accepts the revised rubric or the exchange reaches its configured limit. The longest saved debate we inspected ran 28 messages after the opening audit.

4.4 Human validation

We compare the resulting rubrics with independently collected maintainer criteria on a reviewed subset. Separately, we compare the Pass or Fail verdicts of a scoring model with human judgments. The first check asks whether the rubric captures the requirements that maintainers consider important. The second asks whether the scoring model applies those requirements correctly.

On 42 maintainer-reviewed items, the automatic documentation-need gate agrees with the maintainers on 41, with one false positive. On 21 documentation-needed items with completed criterion alignment, mean priority-weighted recall against maintainer criteria is 0.892. In a separate study of 20 items and 330 criteria, the scoring model agrees with the post-adjudication human reference on 310 criteria (93.9%). One paper author resolved disagreements in the human reference after review, so this figure is not blinded agreement between two humans. Appendix B describes both validation studies.

5 Experimental Setup

We evaluate seven agent lanes:

- GLM 5.2, Qwen3.8 Max, and Kimi K2.7 Code with OpenCode
- Claude Opus 4.8 and Claude Sonnet 4.6 with Claude Code
- GPT-5.5 and GPT-5.6 Sol with Codex

Every agent runs each item in a fresh Docker environment with the same inputs. The inputs are the pre-change code and documentation repository, plus the trigger: a code diff or a reported documentation gap. Each agent-item pair runs once and must either return a patch or abstain. Agents can use command-line tools to inspect and edit the repository, but the environment has no network access except to the model providers.

Why the environment is sealed

Internet access can be valuable for documentation agents in ordinary use, but DOGBENCH blocks it to prevent contamination. Because the source events and merged documentation are public, a connected agent could retrieve the answer instead of solving the task from the supplied evidence. In an audit of an earlier version of the benchmark, we found that 31% of runs retrieved the source pull request despite identity masking. A further 6% copied directly from other agents’ earlier trajectories because the agents shared a host.

Data splits

The release has two splits. The development split holds 175 items: 123 documentation-needed and 52 abstention. The held-out split holds 117 items: 82 documentation-needed and 35 abstention. The random split draw comes from a procedure stratified by class, source pool, task type, and repository. We did not choose the split by inspecting measured scores. Development items include inputs, rubrics, and reference artifacts. Held-out rubrics, references, and item-level scores remain private. We report only the seven reproducible agents in this paper.

6 Results

Table 1 reports the composite score on the 117-item held-out split. The composite score combines two capabilities: delivering useful patches when documentation is needed and correctly refraining from editing when it is not. The table reports the following measures:

Delivered quality.

Delivered patch quality, D , averages rubric scores across the 82 held-out items that require an update. A missed or empty patch scores zero. Conditional quality averages only valid emitted patches.

Abstention recall.

Abstention recall, N , measures correct abstention across the 35 held-out items that need no update.

Composite score.

We combine D and N with the harmonic mean, $C = 2DN/(D + N)$. The harmonic mean treats useful patches and correct abstention as jointly necessary. It also keeps the score independent of the benchmark’s constructed class proportions.

Accuracy.

Decision accuracy covers all 117 items.

P0-clean delivery.

P0-clean delivery is the share of the 82 documentation-needed items that received a valid P0-clean patch (Section 4).

Qwen3.8 Max+OpenCode has the highest composite score (47.3), followed by GPT-5.6 Sol+Codex (46.2) and GLM 5.2+OpenCode (44.2). GPT-5.6 Sol+Codex has the highest P0-clean delivery (39.0). Appendix C reports results on all 292 items separately.

Agent	Score	Accuracy	Patch recall	Abstention recall	P0-clean delivery	Delivered quality	Conditional quality
Qwen3.8 Max							
+OpenCode	47.3	79.5	80.5	77.1	26.8	34.1	42.3
GPT-5.6 Sol							
+Codex	46.2	81.2	95.1	48.6	39.0	44.1	46.3
GLM 5.2							
+OpenCode	44.2	83.8	84.1	82.9	23.2	30.1	35.8
Kimi K2.7 Code							
+OpenCode	43.9	82.1	91.5	60.0	24.4	34.6	37.8
Claude Opus 4.8							
+Claude Code	41.2	81.2	79.3	85.7	24.4	27.1	34.2
Claude Sonnet 4.6							
+Claude Code	40.0	82.9	93.9	57.1	20.7	30.8	32.8
GPT-5.5							
+Codex	34.0	76.1	96.3	28.6	36.6	42.0	43.6

Table 1: Primary results on the 117-item held-out split (%). Rows are ordered by the unrounded composite score.

6.1 Patch or abstention decision performance

The held-out decision task contains 82 documentation-needed items (70.1%) and 35 abstention items (29.9%). Table 2 treats `patch` as the positive class.

Agent	Accuracy	Patch recall	Abstention recall
GLM 5.2+OpenCode	83.8 (98/117)	84.1 (69/82)	82.9 (29/35)
Claude Sonnet 4.6+Claude Code	82.9 (97/117)	93.9 (77/82)	57.1 (20/35)
Kimi K2.7 Code+OpenCode	82.1 (96/117)	91.5 (75/82)	60.0 (21/35)
GPT-5.6 Sol+Codex	81.2 (95/117)	95.1 (78/82)	48.6 (17/35)
Claude Opus 4.8+Claude Code	81.2 (95/117)	79.3 (65/82)	85.7 (30/35)
Qwen3.8 Max+OpenCode	79.5 (93/117)	80.5 (66/82)	77.1 (27/35)
GPT-5.5+Codex	76.1 (89/117)	96.3 (79/82)	28.6 (10/35)

Table 2: Patch or abstention decisions for seven lanes on the 117-item held-out split (%). Parentheses give the numerator and denominator. Patch recall measures recovery of required updates, and abstention recall measures correct abstention.

Abstention recall ranges from 28.6% to 85.7%. GLM 5.2+OpenCode has the highest held-out decision accuracy (83.8%). GPT-5.5+Codex recovers 96.3% of required patches, and GPT-5.6 Sol+Codex recovers 95.1%. They make 25 and 18 incorrect decisions, respectively, on the 35 abstention items.

Unnecessary edits have real cost for both the documentation reader and the maintainers. These edits can add implementation details that users neither need nor can act on, which bloats the documentation and makes relevant guidance harder to find. Every unnecessary patch also needs maintainer attention during triage, review, and ongoing maintenance, and it adds work to downstream tasks such as translation and versioning. Abstaining from an unwarranted edit is therefore a documentation-quality and governance requirement, and we think the benchmark should measure it.

6.2 Documentation quality results

Table 3 reports quality conditional on a correct patch decision. GPT-5.6 Sol+Codex (46.3) and GPT-5.5+Codex (43.6) have the highest means.

Agent	Mean	95% CI	Median	Correct patches (<i>n</i>)
GPT-5.6 Sol+Codex	46.3	[40.8, 51.3]	50.0	78
GPT-5.5+Codex	43.6	[37.8, 48.7]	42.9	79
Qwen3.8 Max+OpenCode	42.3	[35.0, 48.7]	42.8	66
Kimi K2.7 Code+OpenCode	37.8	[31.9, 43.3]	35.7	75
GLM 5.2+OpenCode	35.8	[29.7, 41.8]	35.7	69
Claude Opus 4.8+Claude Code	34.2	[28.5, 40.1]	36.4	65
Claude Sonnet 4.6+Claude Code	32.8	[27.7, 38.1]	33.3	77

Table 3: Scored documentation quality on the 82 documentation-needed held-out items, conditional on a correct patch decision. Empty outputs, abstentions, and wrong decisions receive no quality score here. Their cost appears in delivered patch quality and the composite score. Intervals are percentile 95% intervals from 20,000 bootstrap resamples of documentation repositories.

6.3 Ablations

Source-event-date ablation We use source-event dates to probe training-time exposure: pull-request merge dates and issue creation dates. For each agent whose model has a provider-published knowledge cutoff, we divide all 205 documentation-needed items at the cutoff. All four agents score lower on post-cutoff items, but every repository-clustered interval includes zero, so the comparison provides no statistically conclusive evidence of a cutoff effect.

Agent	Pre n	Pre score	Post n	Post score	Δ	Repo 95% CI
Claude Opus 4.8+Claude Code	86	29.4	119	27.1	-2.2	[-9.5, 4.8]
Claude Sonnet 4.6+Claude Code	50	34.6	155	30.9	-3.7	[-12.8, 5.3]
GPT-5.5+Codex	69	42.9	136	40.9	-2.0	[-9.0, 7.1]
GPT-5.6 Sol+Codex	94	44.3	111	44.0	-0.3	[-6.6, 6.7]

Table 4: Mean documentation-quality score before and after each model’s published knowledge cutoff, on all 205 documentation-needed items. Event dates are pull-request merge dates or issue creation dates. Δ is the post-cutoff score minus the pre-cutoff score, and intervals resample repositories.

Repository-footprint ablation We also compare 82 items from low-footprint repositories (fewer than 1,000 GitHub stars) with 91 items from popular repositories (over 5,000 GitHub stars). The comparison shows no consistent advantage for popular repositories. Table 16 in the appendix gives per-agent values.

Documentation-scale ablation We also tested whether larger documentation sets affect scores. Across repository-level means, doubling the page count changes the score by -0.98 points (95% CI -2.81 to $+0.69$). The interval includes zero, so the data show no clear relationship between documentation size and score.

7 Failure Analysis

7.1 Patch/abstention decision failures

Agents make decision failures by making edits when no change is needed (*over-editing*) and by failing to edit when change is needed (*under-editing*).

Over-editing often starts with a misleading cue. The cue is an internal symbol with a user-facing-sounding name, or an existing documentation page that mentions the affected component. The agent treats that cue as proof that the change alters documented user behavior. Typically, the agent documents one of the following, none of which needs documentation:

- an internal refactor, rename, or mechanically regenerated type whose public contract is unchanged
- a performance optimization, test, CI, or dependency change with no observable user effect
- generated files or reference artifact that should not be manually updated

These patches often look defensible because the terms and destination are topically related to the diff. The missing step is establishing that the change creates behavior a user can invoke, observe, or act on. Once an agent predicts that it should edit, it tends to search for somewhere to put text. It does not go back to ask whether a documentation obligation exists, even when it later finds evidence for the opposite decision.

Under-editing happens when agents treat implementation location, the absence of existing coverage, or weak keyword overlap as evidence that no documentation change is needed. These proxies cause agents to miss user-facing changes. Common forms include the following:

- treating a real public interface, such as a SQL keyword, API function, configuration option, or credential setting, as an implementation detail because it lives in a parser, registry, or configuration module
- reading a new data source, integration, or landing-page capability as internal pipeline plumbing rather than a change to the reader’s available workflow
- treating a failed search for existing coverage as evidence that a subject is intentionally undocumented, when the missing coverage may be the documentation gap the task exposes

- declining documentation-only or issue-driven work because there is no implementation diff, even when the task identifies a verified gap in existing behavior

The *absence-as-evidence* case reinforces itself. In incorrect-abstention trajectories, an agent searches the existing documentation for the feature, interface, or workflow that the task affects. It finds no coverage and reads that silence as evidence that the subject is intentionally out of scope. From the agent’s perspective, deliberate exclusion and an unfilled gap produce the same search result. Once the agent treats absence as policy, the omission propagates.

7.2 Documentation quality failures

Technical-writing failure mode	Effect on the documentation	Rate
Task-completion gap	Omits a decision point, procedure, verification step, or recovery path needed to complete the reader’s task.	45.5%
Technical inaccuracy	Misstates an interface or behavior, including its scope, default, lifecycle, or compatibility.	36.6%
Incomplete conceptual or reference coverage	Omits the core concept, capability, or contract the reader needs to understand and use the change.	32.5%
Missing supporting information	Covers the main task but omits material rationale, boundaries, examples, operational detail, or secondary cases.	26.5%
Missing prerequisites	Omits permissions, dependencies, credentials, versions, resources, enablement, or other setup conditions.	21.0%
Information-architecture or findability failure	Places content outside the reader’s likely path or omits navigation, cross-references, and findable terminology.	20.7%
Missing audience and purpose framing	Does not establish who the content is for, why it matters, or when to use it.	19.9%
Cross-surface content inconsistency	Updates one surface while leaving an authoritative, mirrored, generated, or linked surface stale.	13.3%

Table 5: Share of submissions with each patch-level problem, across the 1,267 submissions audited before the reruns. The table lists labels assigned to at least 10% of submissions, and Table 17 in the appendix lists the rest. One patch can have several problems. The labels describe defects in the patch itself. Table 6 covers the causes in the agents’ trajectories. Only 1.3% of submissions were labeled “no material defect” among the selected top labels.

Hallucinations often distort real behavior

Our audit labeled only 6.1% of submissions as containing fabricated content, a category that includes invented classes, flags, and endpoints. Technical inaccuracies appeared in 36.6% of submissions and involved scope, defaults, lifecycle, or compatibility. The failure taxonomy classifies invented interfaces or capabilities as fabricated content and false descriptions of real interfaces as technical inaccuracies. In practice, the boundary is not always clear.

One agent wrote that Helm 4 uses Server-Side Apply by default when installing or upgrading releases.¹ That default applies to new installations. Releases created with Helm 3 continue using client-side apply after upgrading unless the user explicitly switches them. The agent explained this distinction later in the patch, but its opening statement still gave readers the wrong default. We classify this error as a technical inaccuracy because the feature exists. Extending the feature’s behavior beyond its supported conditions could also reasonably count as hallucination. In many audited cases, the agent took behavior that holds under narrow conditions and presented it as true in general. These claims are unfounded, like hallucinations, but they distort real functionality instead of inventing it.

¹Item `helm-helm-www-pr1926`, Claude Sonnet 4.6+Claude Code.

Agent patches often lack a model of the reader’s task

Agents can often describe a feature’s behavior. They are less able to write for a reader who came to the documentation to decide something or reach a goal. Audience and purpose framing is missing in 19.9% of audited submissions. These patches explain what a feature does but not who should use it, why it is useful, or when to choose it. For example, one Strawberry GraphQL patch correctly documented how to select an older Apollo Federation version. It did not explain why a reader might need to: to upgrade Strawberry while staying compatible with an older Apollo Router or Gateway.² The patch documented the setting but omitted the decision it was designed to support.

This limitation extends beyond explaining when or why to use a feature. Much technical documentation guides readers through a task, and the reader’s goal is to complete that task. Doing so may require prerequisites, intermediate decisions, procedural steps, verification, and recovery guidance. Task-completion gaps appear in 45.5% of audited submissions, and missing prerequisites in 21.0%. These failures suggest that agents treat a change as one piece of information to convey rather than as one part of a larger user journey. A patch may therefore describe the behavior accurately and still leave the reader unable to accomplish the task that brought them to the documentation.

This narrow view also affects how agents treat the documentation as a whole. Readers, both humans and agents, reach a page through search, navigation, related guides, and examples. Agents may add accurate information to a page that the intended reader is unlikely to visit, create a page without linking it from the relevant workflow, or update one surface and leave another surface on the same topic stale. Information-architecture or findability failures appear in 20.7% of audited submissions, and cross-surface inconsistencies in 13.3%. A patch can therefore be accurate in isolation and still fail within the larger documentation system.

7.3 Trajectory analysis of failure root causes

Patch-level labels describe what is wrong with the resulting documentation, but not why the agent produced it. We therefore inspected the trajectory behind each of the 1,267 audited submissions. For each material problem, we assigned one or more causes that the trace supports. Table 6 reports submission-level rates across this full population.

²Item strawberry-graphql-strawberry-pr4045, GPT-5.5+Codex.

Trajectory-level root cause	Observable reasoning failure	Submissions	Rate
Stopped at explaining the interface without examining how it is used in practice	The agent described changed fields, settings, callbacks, or lifecycle mechanics, but did not test the explanation against the reader’s setup, decision, execution, verification, or recovery path.	456	36.0%
Did not inspect decisive evidence and filled the gap with a plausible assumption	The agent found related material but stopped before the controlling implementation, schema, test, or public contract, then completed the explanation with a convention that sounded reasonable.	420	33.1%
Stopped searching after finding the first plausible documentation surface	The agent found a reasonable page to edit and did not continue checking other maintained, generated, mirrored, migration, or workflow surfaces affected by the same change.	382	30.1%
Inspected relevant evidence but did not convert it into a complete coverage checklist	The agent reached evidence bearing on the requirement but began drafting without tracking the claims, setup, boundaries, examples, and reader actions that needed to survive into the final patch.	349	27.5%
Committed too early to a narrow interpretation of the task	Before completing the investigation, the agent declared the task to be a rename, reference update, single-page edit, or similarly narrow deliverable and ignored evidence outside that frame.	346	27.3%
Overgeneralized or misinterpreted partial evidence	The agent inspected relevant evidence but converted one branch, example, implementation detail, or deployment pattern into a broader or different public rule.	333	26.3%

Table 6: The six most common trajectory-level root causes across the 1,267 pre-rerun trajectories, which were frozen separately. Multiple causes may apply, so rates do not sum to 100%. Table 18 in the appendix lists the less common causes.

Agents often lack a reliable test for whether they have gathered enough evidence. Sometimes agents stop researching before they reach the decisive evidence. Other times they begin drafting from partial information without recognizing that their evidence is incomplete. This pattern suggests a failure to recognize uncertainty. Prior work reports similar findings [8, 11, 18]. Models struggle to identify the source of uncertainty. Information-seeking agents often answer before the available evidence is sufficient, and they do not reliably recognize when more information gathering has value.

Premature closure, shifting from investigation to drafting too soon, cuts across many of the root causes. Once an agent finds a plausible interpretation or a reasonable page to edit, it often starts drafting. It may stop before reaching key evidence, and it may also stop before checking every affected documentation surface, which leaves parts of the documentation stale. Insufficient search is only part of the problem. The larger part is that agents lack a reliable stopping rule. Such a rule would tell an agent when it understands the task, the evidence, and the documentation impact well enough to begin writing.

We found no clear relationship between the assigned root cause and trajectory length, whether measured by turns or by token use. Effort also did not rise with task difficulty. We defined an item’s difficulty from the mergeability of the other six agents’ patches on that item. Within each agent, the rank correlation between difficulty and effort was +0.034 for processed tokens, +0.034 for trace-event count, and +0.030 for tool actions. All task-clustered 95% confidence intervals include zero. Premature closure therefore does not necessarily produce a short trajectory. An agent may stop investigating early and then spend substantial effort drafting, revising, or elaborating an incomplete account.

8 Discussion

8.1 Missing context about the reader

Some failures that we attribute to a missing model of the reader may instead reflect missing context about how the software is used. Agents cannot always infer from parametric knowledge alone what readers are

trying to accomplish or which details they need. That inference is especially hard when user motivations and the surrounding workflow context are implicit rather than stated.

8.2 Coarse training rewards

Premature-closure failures may be related to coarse reward signals during post-training. RAGEN finds that trajectory-level rewards do not reliably teach agents how to reason through multi-turn tasks. Without fine-grained, reasoning-aware feedback, agents may learn shallow strategies or produce reasoning that is not grounded in the environment [20]. Kim et al. report a similar pattern [6]. In their experiments, outcome-only reinforcement learning improved final accuracy but made intermediate reasoning less accurate and less internally consistent. Models learned shortcuts rather than reliable reasoning procedures. These results offer possible explanations for our findings. The agents seemed to infer scope from early cues, such as the location of a code change or the name of a feature. They then began drafting within that narrow frame and filled evidence gaps with plausible assumptions that their environment did not support.

8.3 Knowing when to stop investigating

Another explanation is that deciding when the evidence is sufficient is itself a difficult capability. SeekBench reports that search agents trained with reinforcement learning answered before gathering sufficient evidence in 76.5% of the evaluated trajectories [18]. CaRT shows that models may rely on superficial stopping rules, such as the number of turns, instead of checking for a decisive fact [8]. Related studies report that language models struggle to retract an earlier inference when new evidence contradicts it and tend to seek examples that confirm an initial hypothesis rather than examples that might disprove it [5, 21]. These findings match the patterns in our trajectory analysis.

8.4 Additional guidance and scaffolding

Several changes could plausibly address the observed failures: explicit instructions in the prompts that ask agents to consider the reader’s goal, skills that emphasize task completion and findability, broader tools, scratch notes, and explicit verification. We explored these approaches informally but did not systematically compare them against a baseline, so we cannot conclude whether they improved documentation quality. Future work should test their effects through controlled comparisons.

9 Limitations

9.1 Measurement

Task-specific rubrics and LLM judges (Section 4) let us score open-ended documentation patches at scale. Because human validation covers only part of the evaluation, automated rubric generation and scoring may still introduce errors. We check commands and examples against the available evidence instead of running every documented procedure in its repository’s native build and runtime environment. As a result, the benchmark has no deterministic checks for code samples and links.

We find that generated rubrics tend to contain more criteria than maintainer-authored rubrics. Many of these additional criteria identify valid documentation improvements, but maintainers may consider them less important. During human calibration, reviewers prioritized the noncritical P1–P3 criteria differently. Depending on repository norms, some emphasized style, while others placed less weight on it. Some preferred comprehensive documentation, whereas others favored a simple, easy-to-follow user path over

broadier coverage. These preferences do not support a single universal weighting of P1–P3 criteria. The score therefore weights all criteria equally in the mean and handles P0 failures separately through the score cap. The published dataset retains the P0–P3 labels, and the accompanying scoring code lets practitioners apply other weights.

Manual review found that some rubric criteria overlap and are not fully independent. Some overlap is warranted, because a single documentation failure can cause several related problems. In a later audit, we tried to merge overlapping criteria. Merging sometimes lost important distinctions, so we kept the overlapping criteria.

9.2 No internet access

We evaluated agents without internet access (Sections 3.2 and 5). To assess how this restriction affected performance, we reviewed 683 trajectories from items on which no agent produced a mergeable patch. We found blocked network requests in 73 runs (10.7%). In 63 of these runs, network access was not necessary to produce a correct patch. The requests mostly involved setup or validation, such as installing dependencies, building documentation, running formatters or linters, and parsing YAML or JSON configuration files. Only 10 runs (1.5% of all audited trajectories) tried to retrieve external evidence that a correct patch required and the supplied inputs lacked.

Internet access might therefore have helped on a small number of items. However, in an earlier web-enabled pilot, 26 of 85 runs (30.6%) retrieved the item’s upstream pull request and its merged documentation. Given this contamination risk, we kept the reported evaluation sealed.

9.3 Asymmetric label construction

The evidence for the abstention and patch labels (Section 3.1) is not equally strong. Documentation-needed items often have direct evidence. Some abstention labels, by contrast, rely only on the absence of a related documentation change within a 90-day audit window. That absence does not necessarily show that documentation was unnecessary. An update may have been forgotten, or it may have happened after 90 days without a link to the code pull request. As a result, some items that needed documentation may be mislabeled as abstention items. This asymmetric label noise could distort the measured decision performance.

9.4 Single-run evaluation

For budget and time reasons, we run each agent on each item once (Section 5). We therefore do not report $\text{pass}@k$, pass^k , best-of- k performance, or within-item run-to-run variance. The results characterize one sampled trajectory per agent and item. They do not measure the probability that an agent reliably produces the same decision or documentation quality across repeated attempts.

9.5 Low-information prose may be under-penalized

The rubric-based scoring (Section 4) and the patch-level failure rates (Section 7.2) may miss low-information prose. A general instruction to identify coherent but low-information prose flagged 3.2% of submissions. A second prompt asked the judge to flag submissions with two or more specific patterns. The patterns included redundant paraphrases, unnecessary explanations, excessive bulleted lists, formulaic contrasts (not X, but Y), three-part constructions, and heavy use of em dashes. This prompt flagged 7.6% of submissions. The increase suggests that LLM judges may miss low-information prose during scoring.

Disclosure

Two authors are affiliated with Promptless, a company that builds documentation agents.

10 Conclusion

DOGBENCH shows that even frontier models paired with frontier coding harnesses cannot yet reliably produce expert-level user-facing documentation in one attempt. The highest composite score on the 117-item held-out split is 47.3 out of 100. Agents still misjudge whether documentation is needed, and when they do edit, their patches may be factually correct but miss what readers need. Fluent prose and capable repository tooling do not yet close this gap.

DOGBENCH measures this gap with 292 real items, drawn from software changes and reported documentation gaps, and it shows where decisions and patches fail. We release the evaluation harness, item schema, dataset card, and development examples so that others can build on the benchmark. Benchmark materials and release information are available at <https://dogbench.ai>.

References

- [1] Choudhury, S. Process reward models for LLM agents: Practical framework and directions. *arXiv preprint arXiv:2502.10325*, 2025.
- [2] Gao, L., Schulman, J., and Hilton, J. Scaling laws for reward model overoptimization. *ICML*, 2023.
- [3] Husain, H., Wu, H.-H., Gazit, T., Allamanis, M., and Brockschmidt, M. CodeSearchNet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*, 2019.
- [4] Jimenez, C. E., Yang, J., Wettig, A., et al. SWE-bench: Can language models resolve real-world GitHub issues? *ICLR*, 2024.
- [5] Jhaveri, A. R., GX-Chen, A., Sucholutsky, I., and Choi, E. Failing to falsify: Evaluating and mitigating confirmation bias in language models. *arXiv preprint arXiv:2604.02485*, 2026.
- [6] Kim, K., Wang, K., Xie, Y., et al. Correct answers from sound reasoning: Verifiable process supervision for language models. *COLM*, 2026.
- [7] Lightman, H., Kosaraju, V., Burda, Y., et al. Let’s verify step by step. *arXiv preprint arXiv:2305.20050*, 2023.
- [8] Liu, G., Qu, Y., Schneider, J., Singh, A., and Kumar, A. CaRT: Teaching LLM agents to know when they know enough. *arXiv preprint arXiv:2510.08517*, 2025.
- [9] OpenAI. Why SWE-bench Verified no longer measures frontier coding capabilities. <https://openai.com/index/why-we-no-longer-evaluate-swe-bench-verified/>, 2026.
- [10] Lu, S., Guo, D., Ren, S., et al. CodeXGLUE: A machine learning benchmark dataset for code understanding and generation. *NeurIPS Datasets and Benchmarks*, 2021.
- [11] Liu, J., Peng, J., Wu, X., et al. Do not abstain! Identify and solve the uncertainty. *ACL*, pages 17177–17197, 2025.

- [12] Nguyen Hoang, A., Le-Anh, M., Le, B., and Bui, N. D. Q. CodeWiki: Evaluating AI’s ability to generate holistic documentation for large-scale codebases. *Findings of ACL*, pages 5812–5827, 2026.
- [13] Mei, W., Gu, Z., Bai, Z., et al. Deep Research as Rubric for Reinforcement Learning. *arXiv preprint arXiv:2606.01091*, 2026.
- [14] Panickssery, A., Bowman, S. R., and Feng, S. LLM evaluators recognize and favor their own generations. *NeurIPS*, 2024.
- [15] Pai, K., Devanbu, P., and Ahmed, T. CoDocBench: A dataset for code-documentation alignment in software maintenance. *arXiv preprint arXiv:2502.00519*, 2025.
- [16] Sainz, O., Campos, J. A., García-Ferrero, I., et al. NLP evaluation in trouble: On the need to measure LLM data contamination for each benchmark. *EMNLP Findings*, 2023.
- [17] Shankar, S., Zamfirescu-Pereira, J. D., Hartmann, B., et al. Who validates the validators? Aligning LLM-assisted evaluation of LLM outputs with human preferences. *UIST*, 2024.
- [18] Shao, J., Lin, Y., Lohani, M. P., Miao, Y., and Luo, B. Do LLM agents know how to ground, recover, and assess? A benchmark for epistemic competence in information-seeking agents. *arXiv preprint arXiv:2509.22391*, 2025.
- [19] Wang, X., Hu, R., Gao, C., Gao, P., and Peng, C. Evaluating repository-level software documentation via question answering and feature-driven development. *arXiv preprint arXiv:2604.06793*, 2026.
- [20] Wang, Z., Wang, K., Wang, Q., et al. RAGEN: Understanding self-evolution in LLM agents via multi-turn reinforcement learning. *arXiv preprint arXiv:2504.20073*, 2025.
- [21] Wilie, B., Cahyawijaya, S., Ishii, E., He, J., and Fung, P. Belief revision: The adaptability of large language models reasoning. *EMNLP*, pages 10480–10496, 2024.
- [22] Ye, Z., Shi, W., Liu, Y., et al. Look before you leap: Autonomous exploration for LLM agents. *arXiv preprint arXiv:2605.16143*, 2026.
- [23] Zhou, H., Huang, H., Long, Y., et al. Mitigating the bias of large language model evaluation. *Proceedings of the 23rd Chinese National Conference on Computational Linguistics*, pages 1310–1319, 2024.
- [24] Zhou, J., Zhang, Q., Wang, Y., et al. RubricBench: Aligning model-generated rubrics with human standards. *ACL*, pages 31179–31200, 2026.

A Dataset Construction and Task Examples

A.1 Additional dataset details

For documentation-needed pull requests, we also require 5 to 500 changed lines of user-facing documentation across files. The lower bound is meant to exclude small, mechanical edits. The upper bound is meant to exclude broad rewrites and changes to the hosting framework.

Empirically, about six routine pull requests need no documentation update for every one that does. The benchmark’s constructed class proportions (87 abstention items and 205 documentation-needed items) are therefore not a prevalence estimate.

A.2 Task, patch, and evaluation case studies

Table 7: Pants: GLM 5.2+OpenCode documents coverage merging, and the saved evaluation passes all P0 criteria.

Setting

Item: pantsbuild-pants-pr23219. **Agent:** GLM 5.2+OpenCode. **Source:** [Pants PR #23219](#). This is a documentation-only item, so the agent receives no implementation code diff. An update is required.

Task request supplied to the agent

Document combining Python coverage from sharded Pants test runs and applying the coverage threshold to the combined result.

Human merged patch (excerpts)

From docs/docs/python/goals/test.mdx. [...] marks omissions.

Incomplete coverage, raw output, and threshold placement

```
+When using the `--shard` flag to split tests across CI runners, each shard only
  exercises a fraction of your test targets. The per-shard coverage report will be
  artificially low. To get accurate coverage you need to combine the binary `.coverage`
  files from all shards.
[...]
+Add `raw` to your coverage reports so Pants writes the `.coverage` binary, and set
  `relative_files = true` so that `coverage combine` can match paths across different
  sandbox directories:
[...]
+:::caution Don't set `fail_under` in `[coverage-py]` when sharding
+Each shard only runs a fraction of your targets, so per-shard coverage is intentionally
  incomplete. Setting `fail_under` in `pants.toml` or `pants.ci.toml` will cause every
  shard to fail. Enforce the threshold after combining all shards instead.
+:::
```

Collection, combination, and reporting

```
+After all shards complete, collect their `.coverage` binaries, combine them with
  `coverage combine`, and generate the final report. The following example uses GitHub
  Actions, but the same approach applies to any CI system:
[...]
+      coverage combine --rcfile=.coveragerc .coverage.shard*
+      coverage xml    --rcfile=.coveragerc -o coverage-report/coverage.xml
+      coverage report --rcfile=.coveragerc --fail-under=80
```

Table 7 continued: Pants coverage merging

Per-shard global_report caveat

```
+:::note `global_report` and sharding
+With `[coverage-py] global_report = true`, per-shard reports show 0% for untouched
  files. Consider applying this setting only in the post-merge step rather than in
  `pants.ci.toml`.
+:::
```

Link from advanced-target-selection.mdx

```
+When using `--shard` with test coverage enabled, each shard only exercises a fraction of
  your targets, producing artificially low coverage numbers. You need to combine the
  coverage data from all shards in a post-shard CI step to get accurate results. To
  learn how to do this for Python see [Coverage with test
  sharding](../python/goals/test.mdx#coverage-with-test-sharding) for the full
  configuration and CI workflow.
```

These excerpts omit the GitHub Actions upload and download steps and other configuration details. The judgments below assess the model-generated patch, not these human excerpts.

Model-generated patch (excerpts)

Each label names the rubric criterion below that the excerpt addresses. Excerpts come from docs/docs/python/goals/test.mdx unless noted.

Why shard coverage must be combined (C1)

```
+Things are different when you shard your tests across multiple CI jobs with `--test-shard
  =k/N` (see [Sharding the input targets](../using-pants/advanced-target-selection.
  mdx#sharding-the-input-targets)). Each shard is a separate `pants test` invocation, so
  each one generates its own coverage report and evaluates `fail_under` against only
  the tests in that shard. The per-shard numbers will usually be lower than the combined
  number, and a passing combined threshold can appear to fail.
+
+To apply `fail_under` to the coverage combined across all shards, do the threshold
  check outside of Pants, on the union of the per-shard data:
```

Raw output and threshold placement (C3)

```
+ [coverage-py]
+ report = ["raw"]
+ # Do not set fail_under here; apply it after combining instead.
```

Preserving shard data, combining, and enforcing the threshold (C3)

```
+ $ pants test --shard=0/3 --use-coverage ::
+ $ cp dist/coverage/python/.coverage .coverage.shard0
+ $ pants test --shard=1/3 --use-coverage ::
+ $ cp dist/coverage/python/.coverage .coverage.shard1
+ $ pants test --shard=2/3 --use-coverage ::
+ $ cp dist/coverage/python/.coverage .coverage.shard2
+ $ coverage combine .coverage.shard0 .coverage.shard1 .coverage.shard2
+ $ coverage report --fail-under=70
```

Link from the sharding guide (C10)

```
+If you use test sharding together with Python coverage and a `[coverage-py].fail_under`
  threshold, note that each shard evaluates the threshold against its own coverage
  only. See [Combining coverage from sharded test
  runs](../python/goals/test.mdx#combining-coverage-from-sharded-test-runs) in the
  Python test docs for how to apply the threshold to the coverage combined across all
  shards.
```

The human patch supplies a GitHub Actions workflow, while the model supplies a generic numbered procedure and shell commands. The rubric does not require matching the human implementation. The full model patch omits the global_report caveat (C15).

Table 7 continued: Pants coverage merging

Selected rubric criteria and verdicts

Full criteria, including pass and fail conditions, are available on the [benchmark website](#).

ID / priority	Verdict	Rubric requirement
C1 / P0	Pass	The patch must explicitly document that separate Python <code>pants test --shard=k/N</code> invocations each measure only their shard's tests, so an individual shard's coverage is incomplete, and accurate project coverage requires combining data from every shard before producing the final report.
C3 / P0	Pass	After reading, a user must know to: 1. Preserve the <code>.coverage</code> data from every shard. 2. Collect and combine all shard data in a post-shard job. 3. Generate the final report only after combination and enforce its threshold on the combined result, not on each incomplete shard through per-shard <code>[coverage-py].fail_under</code> configuration.
C10 / P1	Pass	The maintained sharding guidance must include a nearby, discoverable connection to the detailed Python coverage-and-sharding guidance. At the pinned base, the natural source is <code>docs/docs/using-pants/advanced-target-selection.mdx</code> near <code>## Sharding the input targets</code> ; an equivalent maintained successor surface passes. The connection must resolve to the actual maintained detailed section. Any repository-supported link form, route, or anchor that resolves correctly passes. A relative <code>.mdx</code> link such as <code>../python/goals/test.mdx#coverage-with-test-sharding</code> is the base-tree conventional example, not the only acceptable spelling.
C15 / P2	Fail	The patch must explain that per-shard <code>[coverage-py].global_report = true</code> can report 0% for files untouched by that shard and should not be treated as final project coverage.

Full-patch outcome: correct patch decision; **92.3/100; P0-clean.**

Table 8: Jujutsu: GPT-5.6 Sol+Codex adds the new type but omits conversion semantics and an affected return type.

Setting

Item: `jj-vcs-jj-pr9347`. **Agent:** GPT-5.6 Sol+Codex. **Source:** [Jujutsu PR #9347](#). In this code-triggered item, the change introduces a byte-string template type. The agent receives the code diff and the pre-change documentation and must update `docs/templates.md`.

Agent-visible code diff (excerpts)

Annotation-line content: `cli/src/commit_templater.rs`

```
-         let out_property = self_property.map(|line| line.content);
-         // TODO: Add Bytes or BString template type?
-         Ok(P::wrap_template(out_property.into_template()))
+         Ok(out_property.into_dyn_wrapped())
```

Fallible byte-to-string conversion: `cli/src/template_builder.rs`

```
+         let from_bytes =
+         |s: BString| Ok(String::from_utf8(s.into()).map_err(|err| err.utf8_error())?)
+         ;
+         let property = match self.property.try_into_string() {
+         Ok(string_property) => return Some(string_property),
+         Err(property) => property,
+         };
+         let property = match property.try_into_byte_string() {
+         Ok(bytes_property) => return Some(bytes_property.and_then(from_bytes).
+         into_dyn()),
+         Err(property) => property,
+         };
```

The annotation method stops wrapping its result as a `Template`. The conversion path uses `String::from_utf8` and propagates invalid-UTF-8 errors. These changes motivate the return-type and conversion documentation.

Table 8 continued: Jujutsu byte strings

Human merged patch (excerpts from docs/templates.md)

Annotation-line return type

```
-* `.content() -> Template`: Line content including newline character.  
+* `.content() -> ByteString`: Line content including newline character.
```

Conversion to byte strings

```
+### `ByteStringify` type  
+  
+An expression that can be converted to a `ByteString`.  
+  
+A `String` can be converted to a `ByteString` losslessly. Any types that can be  
+converted to `Template` can also be `ByteStringify`. Unlike `Template`, color  
+labels are stripped.
```

Conversion to UTF-8 strings

```
Any types that can be converted to `Template` can also be `Stringify`. Unlike  
-`Template`, color labels are stripped.  
+`Template`, color labels are stripped. Invalid UTF-8 sequences are rejected.
```

The rest of the human patch introduces `ByteString` and its methods, updates regex-capture return types, and revises the equality documentation.

Model-generated patch (excerpts)

Each label names the rubric criterion below that the excerpt addresses.

New type in the maintained reference (C1, C4)

```
+### `ByteString` type  
+  
+_Conversion: `Boolean`: yes, `Serialize`: yes, `Template`: yes_  
+  
+A byte string can contain arbitrary bytes and does not have to be valid UTF-8.  
+It can be implicitly converted to `Boolean`. When converted to a template, its  
+bytes are preserved without a UTF-8 conversion. The following methods are  
+defined.
```

Conversion definition (C7)

```
+### `ByteStringify` type  
+  
+An expression that can be converted to a `ByteString`. This includes  
+`ByteString` values and any type that can be converted to `Stringify`.
```

Updated regex-capture signatures (C9)

```
-* `.get(index: Integer) -> String`: Returns the capture group at `index`.  
+* `.get(index: Integer) -> ByteString`: Returns the capture group at `index`.  
+  Capture group 0 is the full match. Errors if the index is out of bounds.  
-* `.name(name: Stringify) -> String`: Returns the named capture group `name`.  
+* `.name(name: Stringify) -> ByteString`: Returns the named capture group `name`.
```

The model describes bytes that may not be valid UTF-8 and passes the revised C4. Its conversion definition does not explain formatted template output or label stripping (C7, revised to P1). It updates the two regex-capture signatures but leaves `AnnotationLine.content()` unchanged (C9). It also leaves the `Stringify` section unchanged, so it omits invalid-UTF-8 rejection (C6). The full patch has these omissions, not only the excerpts.

Table 8 continued: Jujutsu byte strings**Selected rubric criteria and verdicts**

We revised C4, C6, and C7 to separate the type definition, UTF-8 rejection, and conversion to bytes. This case reports an evaluation under the revised criteria. Full criteria and pass and fail conditions are available on the [benchmark website](#).

ID / priority	Verdict	Rubric requirement
C1 / P0	Pass	<code>docs/templates.md</code> is updated as the primary reference surface and introduces <code>ByteString</code> as a public template-language type.
C4 / P0	Pass	Explain that <code>ByteString</code> can contain bytes that are not valid UTF-8. A concise statement such as "arbitrary bytes" or "not guaranteed to be valid UTF-8" is sufficient; the phrase "ASCII-compatible" and an explicit comparison sentence with <code>String</code> are not required.
C6 / P0	Fail	Explain that converting a <code>ByteString</code> through <code>Stringify/stringify()</code> requires valid UTF-8 and rejects invalid byte sequences. This is a conversion precondition, not another test of the byte-string definition. A short warning, an error example, or a clear exception to the existing Template-to-Stringify statement is sufficient.
C7 / P1	Fail	Define <code>ByteStringify</code> as accepting values convertible to bytes and explain that strings and formatted template output can supply those bytes without retaining formatting labels. Equivalent descriptions or examples count; the literal word "lossless" is not required. A precise cross-reference to existing conversion documentation can supply these facts. This criterion concerns conversion to bytes, not rejection when converting bytes to UTF-8 strings (C6).
C9 / P0	Fail	The discoverable API reference in <code>docs/templates.md</code> shows all three signatures: <code>AnnotationLine.content() -> ByteString</code> , <code>RegexCaptures.get(index: Integer) -> ByteString</code> , and <code>RegexCaptures.name(name: Stringify) -> ByteString</code> . If the patch edits the global <code>replace(pattern, content, replacement)</code> documentation, it preserves that function's signature.

Full-patch outcome: correct patch decision; **44.4/100; P0 failures C6, C9.**

B Rubric Construction and Human Validation

B.1 Research, audit, and debate

Before adopting rubric-based evaluation, we explored two other approaches. First, we generated questions from the source pull request and checked whether the candidate documentation let a question-answering agent answer them. The questions were often too broad, covering documentation beyond the evaluated patch, or limited to technical details that did not reflect readers' goals. Second, we translated documentation into executable tests. This approach was particularly useful for procedural content, but test outcomes were hard to attribute to the patch under evaluation. A test could fail because of unchanged documentation, environment issues, or assumptions that the test generator introduced. We arrived at the current design through experiments, ablations, and input from human experts.

Research and synthesis

Rubric construction begins with a research agent (GPT-5.6 Sol) that has internet, shell, and browser access. The research agent can install software, run examples, and test behavior to understand the reader's experience. The human-authored patch is not included among the supplied candidate patches, but internet research may uncover it. Every proposed criterion must have independent supporting evidence; a criterion justified only by the human patch is not allowed. The research agent produces a report that covers software behavior, reader needs, affected documentation, dependencies, and relevant constraints.

Audit and debate

During the audit, each disputed criterion needs specific evidence, such as a source-file location, a test, or a maintainer comment. A criterion without that evidence is withdrawn. Each proposed change must also

explain its consequence for the reader. This process can uncover errors in the underlying research or rubric synthesis, as the following example shows.

Example: distinguishing useful detail from required coverage

Doc Detective PR #146 lets users pass multiple input paths in one comma-separated `--input` argument. Criterion C5 concerned whether the documentation should also explain that URL inputs are preserved rather than resolved as local paths (“URL passthrough”). The dispute concerned C5’s priority. Both sides agreed that the behavior exists. The excerpts below come from the saved debate, with omissions marked [. . .].

Auditor, opening audit: lower C5 to P3

Reasoning: URL passthrough is real, but the criterion overstates its importance. The PR body and issue frame the feature as multiple CLI input paths; the tests added for this PR cover local file inputs, not URLs. [...] Revise this from a required P2 criterion into a P3 credit-only quality signal for patches that accurately mention URL values.

Author, first response: retain P2

C5 — REBUT, but partially concede. [...] When the majority of real candidate patches independently document a behavior and omitting it is judged undesirable, that’s evidence it’s a genuine “good-patch” signal, not P3 excellence trivia.

[...] Given the rules say P2 = “a good patch includes it but failing it is still shippable” — that is *exactly* the correct semantics for URL passthrough per both my report and the differential findings. P3 would understate it. [...]

Auditor, second audit: task evidence does not justify P2

I still rebut C5 at P2. [...] The author’s differential evidence can justify it as desirable, but not enough to make it a P2 “good patch includes this” criterion when the issue, PR body, README context, and tests all center local/path multi-input syntax. Also, the current scoring already says a “Good” patch may have C5 partial or failed, which is internally inconsistent with keeping C5 as a required P2.

[...]

Author, second response: concede P3

CONCEDE on C5. [...] Demoting to P3 keeps the credit the differential evidence supports while resolving the internal inconsistency the auditor identified in the scoring guide.

B.2 Human validation for the rubrics and scorer

Project maintainers first submitted free-form, GitHub-style inline review comments on the candidate patches without knowing which patch was which. Claude Sonnet 4.6 used those comments and the lines they annotated to draft structured rubrics. Each maintainer then edited, added, or deleted criteria, approved each criterion individually, and decided whether each candidate patch passed or failed each criterion. This process was separate from automatic rubric generation. Maintainers never saw the corresponding generated rubrics, and the rubric agents never saw the maintainer rubrics. One repository maintainer reviewed each item.

We did not rely only on maintainer rubrics, for two reasons. First, the cost and time were prohibitive for the scope of this study. Constructing one maintainer rubric took roughly 30 minutes to an hour, even for maintainers familiar with the domain. Second, expert review often did not give a consistent or exhaustive

standard across items. Maintainers had different editorial priorities. Some emphasized style, and another emphasized information architecture. One favored self-contained pages, while another preferred progressive disclosure. Each is a defensible choice, but with only one maintainer per repository, those preferences change what the benchmark measures. Recruiting several maintainers for every repository was impractical. Maintainer rubrics can also contain omissions or mistakes, especially when reviewers must anticipate gaps that no candidate patch addresses.

We therefore generated task-specific rubrics from task-specific research and a consistent set of expert-reviewed technical-writing principles. We then validated the generated rubrics against the maintainer rubrics. This approach made evaluation at scale possible while keeping expert judgment central to its design and validation. The validation set contains 42 maintainer-reviewed decisions: 23 items require a documentation change and 19 require no change. The 23 documentation-needed items cover 242 maintainer-validated criteria. Table 9 compares the automatic documentation-need gate with the maintainer decisions.

Subset	n	Acc.	Prec.	Recall	F1	TP	FP	FN	TN
Documentation-need gate	42	0.976	0.958	1.000	0.979	23	1	0	18

Table 9: Agreement between the documentation-need gate and maintainer decisions on the 42 maintainer-reviewed items. Every item has a prediction.

Human-rubric recall	Count	Percentage
Covered	220	90.9%
Missing	17	7.0%
Conflicting	5	2.1%
Total	242	100.0%

Table 10: Coverage of 242 maintainer-validated criteria across 23 documentation-change examples.

The generated rubrics cover 90.9% of 242 maintainer-validated criteria. In the reverse comparison, 59.8% of 361 generated criteria are covered in the maintainer-reference rubrics. Of the 137 criteria absent from the human rubrics, 76 are routine or defensive checks that reviewers often leave implicit. These include prose and markup conventions, repository conventions, and safeguards against fabricated content. Lack of a match therefore does not establish that a generated criterion is invalid. After the human-authored rubrics were complete, experts reviewed the generated rubrics and often agreed with additional criteria they had not identified initially. These follow-up reviews were qualitative and limited in scale.

Generated-rubrics precision	Count	Percentage
Covered	216	59.8%
Missing	137	38.0%
Conflicting	8	2.2%
Total	361	100.0%

Table 11: Overlap precision of the generated rubrics against the maintainer rubrics. Full and partial nonconflicting matches are combined. A partial match does not validate every requirement in a generated criterion. Absence from the maintainer rubric does not show that a criterion is invalid. All conflicts are partial.

We also applied the generated and maintainer rubrics to the same candidate patches. The two sets of scores have a pooled Spearman correlation of 0.755 and an interval Krippendorff α of 0.805. They agree on 85.0% of within-item candidate orderings. These measures suggest that evaluations based on the two kinds of rubric agree.

B.3 Direct criterion-verdict validation

The scorer-validation study covers 20 documentation-needed items and 330 rubric criteria. For each item, GPT-5.6 Terra, Claude Sonnet 5, and a human reviewer judged the same anonymous candidate patch against the same rubric criteria. Requirements and triggered conditional criteria receive Pass or Fail. Deduction-only guardrails are marked violated or not violated. Table 12 reports agreement with the human reference.

Grader	Coverage	Matches	Agreement
GPT-5.6 Terra	20/20	310/330	93.9%
Claude Sonnet 5	20/20	301/330	91.2%

Table 12: Criterion agreement between each grader and the human reference after adjudication, across 20 validation items.

B.4 Remaining P0 failures on human merged patches

Table 13 shows four human merged patches that fail a P0 criterion because of a concrete defect.

Table 13: Four examples of human merged patches with concrete P0 failures. The first column gives the criterion ID and the merged patch’s rubric score out of 100.

Case / criterion / score	Faulty human-patch excerpt	Why the failure remains P0
Pants #22034 C8; 50.0	<code>pants experimental-deploy src/k8s/:webpages</code>	The exact-version address parser rejects the empty path component before the colon, so the deployment command cannot resolve its target. Removing the slash fixes it: <code>src/k8s:webpages</code> . The defect is small but blocks the advertised operation.
OpenCost #102 C6; 44.4	<code>Secret creation: kubectl create secret generic azure-service-key -n kubecost (excerpt). Workload update: helm upgrade opencost . --namespace opencost -f values.yaml</code>	The Secret is created in kubecost, but the workload that mounts it is updated in opencost. A workload cannot mount a Secret from another namespace. The supplied credential-injection procedure therefore fails unless the namespaces are made consistent.

Case / criterion / score	Faulty human-patch excerpt	Why the failure remains P0
Strawberry #4342 C6; 31.2	<pre> Resolver: def create_user(self, email: str) -> str: return email Schema: strawberry.Schema(mutation=Mutation, extensions=[PydanticErrorExtension()],) </pre>	The usage example omits the required query root and never invokes Pydantic validation. It cannot produce the advertised validation errors. The same PR’s working test supplies both a query root and Pydantic model construction, which gives a direct implementation contrast.
dlt #2292 C8; 55.6	<pre> iceberg_tables["my_iceberg_table"] .optimize.compact() </pre>	The helper returns native PyIceberg Table objects. The retained source check for supported version 0.8.1 finds no optimize API or dynamic fallback. The copied Delta-style operation cannot run on that object, so a supported Iceberg operation must replace it.

C Additional Results and Robustness Analyses

C.1 Population-specific robustness results

The paper’s primary comparison uses the 117-item held-out split. Table 14 reports all seven agent lanes on the full 292-item population.

Agent	Score	Accuracy	Patch recall	Abstention recall	P0-clean delivery	Delivered quality	Conditional quality
Qwen3.8 Max+OpenCode	48.1	81.2	84.4	73.6	30.7	35.8	42.4
GLM 5.2+OpenCode	46.9	84.6	84.9	83.9	26.3	32.6	38.4
GPT-5.6 Sol+Codex	45.0	81.2	96.1	46.0	38.5	44.1	45.9
Claude Opus 4.8+Claude Code	42.2	80.8	79.0	85.1	21.0	28.1	35.5
Kimi K2.7 Code+OpenCode	42.0	79.1	88.8	56.3	25.9	33.5	37.7
Claude Sonnet 4.6+Claude Code	39.1	81.5	94.6	50.6	25.9	31.8	33.6
GPT-5.5+Codex	32.3	75.0	95.6	26.4	35.6	41.6	43.5

Table 14: Full-population results for seven agent lanes on all 292 items (205 documentation-needed and 87 abstention items), in percent. Definitions match Table 1. Rows are ordered by the unrounded composite score.

Patch or abstention decision errors

Table 15 counts decision errors on all 292 items, with `patch` as the positive class. A false negative (FN) is an abstention on a documentation-needed item. A false positive (FP) is a patch on an abstention item.

Agent	FN	FP
GLM 5.2+OpenCode	31	14
Claude Sonnet 4.6+Claude Code	11	43
Qwen3.8 Max+OpenCode	32	23
GPT-5.6 Sol+Codex	8	47
Claude Opus 4.8+Claude Code	43	13
Kimi K2.7 Code+OpenCode	23	38
GPT-5.5+Codex	9	64

Table 15: Decision error counts for seven agent lanes on the full 292-item population.

Repository-footprint ablation

We measured repository popularity from GitHub on August 18, 2026. The comparison uses all 205 documentation-needed items and the same 82 low-footprint and 91 popular items for every agent (Table 16). It excludes the 32 items from repositories with 1,000 to 4,999 stars.

Agent	Low-footprint	Popular	Δ low-popular	Repo 95% CI
Claude Opus 4.8+Claude Code	26.3	28.5	-2.2	[-8.1, 5.9]
Claude Sonnet 4.6+Claude Code	31.7	29.9	+1.8	[-3.3, 7.2]
GLM 5.2+OpenCode	33.6	29.9	+3.6	[-4.4, 12.9]
GPT-5.5+Codex	45.7	36.0	+9.7	[+0.8, 16.1]
GPT-5.6 Sol+Codex	47.2	42.4	+4.8	[-3.4, 10.9]
Kimi K2.7 Code+OpenCode	34.2	32.5	+1.8	[-5.9, 9.3]
Qwen3.8 Max+OpenCode	35.8	34.2	+1.7	[-6.6, 9.2]

Table 16: Mean documentation-quality score by repository-popularity band for 173 of the 205 documentation-needed items. Low-footprint repositories have fewer than 1,000 stars, and popular repositories have at least 5,000. Intervals cover the low-minus-popular difference in means from 20,000 percentile bootstrap resamples of repositories within each band.

D Failure Taxonomy: Additional Categories

D.1 Additional artifact-level failure categories

Table 17 lists the artifact-level failure categories that Table 5 omits.

Table 17: Additional artifact-level categories, continuing Table 5, with patch-level rates across the 1,267 submissions audited before the reruns. Multiple categories may apply.

Technical-writing failure mode	Effect on the documentation	Rate
Low information density or poor scannability	Repeats information, adds unnecessary structure, or uses disproportionately long prose for the information conveyed.	7.6%
Fabricated content	Invents an interface, command, control, behavior, version requirement, or guarantee; distinct from a false description of a real interface.	6.1%
Nonfunctional example	Supplies a code block, command, configuration, or API example that would fail or teach the wrong call shape.	2.5%
Documentation-system defect	Breaks links, markup, rendering, terminology, or documentation-system conventions.	0.9%

Technical-writing failure mode	Effect on the documentation	Rate
Ambiguous or contradictory guidance	Gives incompatible instructions or leaves a material rule ambiguous within the edited documentation.	0.4%
Scope creep	Edits unrelated files or topics beyond the documentation need; excludes companion edits required for consistency.	0.2%

D.2 Less common trajectory-level causes

Table 18 lists the trajectory-level causes that Table 6 omits.

Table 18: Less common trajectory-level root causes, continuing Table 6, across the 1,267 pre-rerun trajectories, which were frozen separately.

Trajectory-level root cause	Observable reasoning failure	Submissions	Rate
Found the correct fact, then dropped or contradicted it during drafting	The correct distinction appeared in the evidence or reasoning but disappeared, weakened, or reversed in the patch.	81	6.4%
Selected the wrong or conflicting source as authoritative	The agent trusted stale documentation, generated output, an internal representation, or permissive runtime behavior over the maintained public contract.	72	5.7%
Validated presentation or file mechanics, but not the substantive claim	The agent checked syntax, links, formatting, or file existence without validating the underlying command, example, route, or factual statement.	50	3.9%
Did not perform a reader-priority and compression pass	The agent stopped after inserting relevant content without removing repetition, artificial structure, or low-density prose.	22	1.7%
Did not run documentation-specific validation	The agent omitted the relevant link, markup, navigation, generation, spelling, or build check.	9	0.7%
Lost requirements or edit state during a long trajectory	Earlier requirements vanished after extended investigation, rewriting, or scope expansion.	7	0.6%
Did not recover relevant historical context	The agent missed a relevant issue, pull request, release milestone, comment, or design decision.	5	0.4%
Did not maintain a boundary between required work and unrelated expansion	The agent expanded into adjacent topics or files without tying each edit to the reader need.	2	0.2%