

Documentation Generation Benchmark



DoGBench

from Promptless



What AI Agents Get Wrong When Writing Docs



A DoGBench Field Guide

FROM THE DOGBENCH RESEARCH BY FRANCES LIU, MANNY
SILVA, PAIGE CALVERT, AYU ADIATI, AND SARAH SANDERS

What AI Agents Get Wrong When Writing Docs

A DoGBench Field Guide

BASED ON THE RESEARCH BY

Frances Liu, Manny Silva, Paige Calvert, Ayu Adiati, and Sarah Sanders

Published by Promptless



Free online

Read the web edition and download this PDF at dogbench.ai/field-guide/. The paper, leaderboard, and public items are at dogbench.ai.

What AI Agents Get Wrong When Writing Docs: A DoGBench Field Guide. First edition, October 2026.

Data: the DoGBench results published at dogbench.ai in the release of October 2, 2026. Leaderboard figures are from the held-out split of 117 items unless a chart says otherwise.

Disclosure: two of the five DoGBench authors work at Promptless, a company that builds documentation agents. Promptless publishes this guide.

© 2026 Promptless. The DoGBench paper and data are the work of their authors.

Contents

START HERE

The findings at a glance	5
Key terms	6

CHAPTERS

1 Built with the community	7
2 Every agent fell well short of maintainer standards	13
3 How docs agents fall short	20
4 Five lessons for managing an agent	28
5 A complete patch changes two pages	35
6 Turn the findings into review habits	44

REFERENCE

What DoGBench doesn't measure	49
Glossary	50
Contributors	52
How to cite DoGBench	53
About Promptless	53

The findings at a glance

AI agents can write polished documentation, but they often miss what readers need to complete a task. DoGBench (Documentation Generation Benchmark) asks whether agents can meet expert standards for user-facing documentation. We gave seven agents real docs tasks from open source projects. We scored every patch against the criteria documentation maintainers apply in review. A patch is the set of edits an agent proposes to the docs. This guide is for documentation owners who want to put that research to work.

AT A GLANCE

Six numbers to remember

47.3	out of 100, the best agent’s composite score	p. 14
39.0%	of items needing a patch got one with no critical defect (best agent)	p. 14
45.5%	of audited submissions left the reader unable to finish the task	p. 21
37.1%	of no-change items got an unneeded patch, on average	p. 22
33.1%	of audited submissions stopped short of the decisive evidence and guessed	p. 24
6.1%	of audited submissions contained fabricated content	p. 26

Held-out split of 117 items and an audit of 1,267 submissions.

KEY TERMS

Four terms explain the numbers in this guide

01 · PATCH SCORE

A patch scores 100 when it meets every requirement for its item.

An item is one real docs task. Each criterion in its rubric tests one decision a reviewer would make and counts one point, so 3 of 4 scores 75 out of 100. Failing any P0 criterion caps the score at 60. The score is progress toward meeting every requirement, not a comparison with a human expert.

ILLUSTRATIVE RUBRIC		
P0	States the new default	✓
P0	Lists every step	✓
P1	Links the related guide	✓
P2	Shows an example	✗
3 of 4 pass		75

02 · COMPOSITE SCORE

The composite score is the headline number.

It's the harmonic mean of delivered quality, how well an agent's patches score with a missed patch scoring zero, and abstention recall, how often it correctly leaves the docs alone. It stays high only if both are high, so the top agent's 34.1 and 77.1% give 47.3.

Qwen3.8 Max with OpenCode		
Delivered quality		34.1
Abstention recall		77.1%
Composite score		47.3

03 · AGENT

An agent is a model paired with a harness.

The harness is the tool that runs the model and gives it files and commands. The seven agents in the paper use Claude Code, Codex, or OpenCode. Change either part, and you have a different agent.

Model	+	Harness	=	Agent
Three of the seven agents				
Claude Opus 4.8	+	Claude Code		
GPT-5.6 Sol	+	Codex		
Qwen3.8 Max	+	OpenCode		

04 · HELD-OUT SPLIT

The main results use the held-out split.

Its 117 items have rubrics that aren't public. The glossary (page 50) defines these and the guide's other terms.

117 items

Rubrics not public

1



Built with the community

We validated the benchmark with rubrics from expert technical writers who are also official maintainers of their open source projects. Each item asks an agent to do what a maintainer does. It decides whether the docs need a change, and if they do, it writes a patch.

292

items, each a real docs task
from an open source project

42

items reviewed by the
projects' own maintainers

Every item is a real repository event

Each item starts from a pull request that changed the product, or from a documentation gap that someone reported. The projects include Helm, PostHog, Mautic, and Doc Detective. We built the benchmark with maintainers and technical writers from those communities, who reviewed items and wrote criteria.

FIGURE 1.1

Written with maintainers, checked by reviewers

PAPER AUTHORS

Frances Liu	Promptless
Manny Silva	Promptless and Doc Detective
Paige Calvert	Helm
Ayu Adiati	Mautic
Sarah Sanders	PostHog

COMMUNITY REVIEWERS

Dachary Carey
Fabrizio Ferri-Benedetti
CT Smith

The agent decides first, then writes

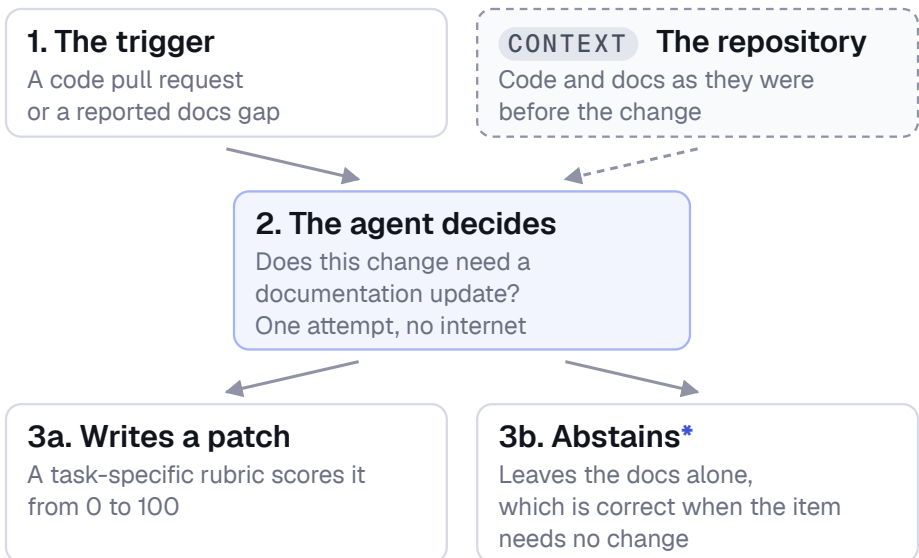
Each agent receives the repository as it existed before the change, plus a triggering signal, such as a code pull request or a reported documentation gap. It doesn't see the docs that

maintainers later merged. It then either writes a patch or abstains, leaving the docs alone. A patch can create new docs, update existing guidance, or delete stale or redundant content.

Agents get one attempt with no internet access. Each agent runs in a fresh, sealed container that can reach only its model provider, so it can't find the merged answer online. In an earlier version of the benchmark, agents found the source pull request online in 31% of their attempts, even though we'd masked the repositories' identities.

FIGURE 1.2

One task, two possible right answers



* Abstaining on an item that needs a patch scores zero for that item.

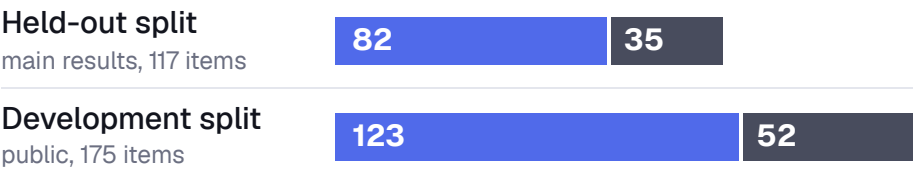
Sometimes the right answer is to leave the docs alone. Of the 292 items, 205 need a patch, and the other 87, the no-change items, need none. A routine refactor, a test change, or a dependency upgrade usually doesn't affect readers. A good maintainer writes nothing in those cases, and neither should an agent. Earlier documentation benchmarks assume someone

already decided an update is needed. We split the items at random into a held-out split for the main results and a public development split. Each split keeps the same mix of answer type, source, task type, and repository.

FIGURE 1.3

Most items need a patch, and some need none

■ Needs a patch ■ Needs no change



Every item that needs a patch has its own rubric

Across the 205 items that need a patch, the rubrics hold 3,273 criteria. An AI agent researches the triggering change and the repository, drafts each rubric, and then compares anonymous candidate patches to find what the draft misses. A second model, from a different model family, audits the result. Each criterion tests one decision a reviewer would make, such as whether the patch states a default correctly. Each also has a priority from P0, the most critical, to P3. The 798 P0 criteria would each block a patch in review on its own, and the P1 to P3 criteria are secondary criteria.

The rubrics check what the reader needs. They don't reward matching the docs the maintainers merged, because many different edits can meet the same need.

The rubrics held up against human judgment. Maintainers reviewed 42 items. On 41 of them, they agreed with the benchmark about whether the docs needed to change. On 23

items that need a patch, maintainers validated 242 criteria of their own, and the generated rubrics covered 90.9% of them.

FIGURE 1.4

Part of one real rubric

PANTSBUILD/PANTS #23219 · TASK-SPECIFIC RUBRIC

- C1 · P0** Explain why per-shard coverage is incomplete
- C2 · P0** State why and when readers need this workflow
- C3 · P0** Give the reader the complete operational sequence
- C4 · P0** Put the primary guidance in maintained Python test coverage documentation
- C5 · P0** Require the `raw` coverage report correctly

Plus 10 more criteria at P0 to P3, including guardrails against invented behavior. Failing any P0 criterion caps the patch's score at 60.

The item asks for docs on combining test coverage reports from split test runs. Chapter 5 follows it from start to finish.

FIGURE 1.5

Three checks against people

41 of 42

decisions on whether docs needed a change matched the maintainers

90.9%

of the 242 criteria maintainers validated appeared in the generated rubrics, across 23 items

93.9%

agreement between the scoring model and the human reference, 310 of 330 criteria

A separate scoring model, GPT-5.6 Terra, checks each patch against each criterion. Its verdicts matched human reviewers' verdicts on 310 of 330 criteria (93.9%).

Strengths can't fully make up for a weakness

Failing one PO criterion caps a patch score at 60, so many small wins can't average away a critical defect.

The composite score is the harmonic mean of delivered quality and abstention recall. Delivered quality is the average patch score on the 82 held-out items that need a patch. A missed patch, an item that gets no patch at all, scores zero. Abstention recall is the share of the 35 no-change items on which the agent leaves the docs alone. A harmonic mean stays low when either side is low.

FIGURE 1.6

Doing one job well is not enough

	Delivered quality		Abstention recall		Composite
GPT-5.5 Codex		42.0		28.6%	34.0
Qwen3.8 Max OpenCode		34.1		77.1%	47.3
Claude Opus 4.8 Claude Code		27.1		85.7%	41.2

Composite = $2 \times \text{delivered quality} \times \text{abstention recall} \div (\text{delivered quality} + \text{abstention recall})$

Held-out split.

2



Every agent fell well short of maintainer standards

The seven agents in the paper, each a model paired with a harness, all scored far from a full score. On most items that needed a patch, even the best agent didn't deliver one free of critical defects.

47.3

the best composite score,
out of 100

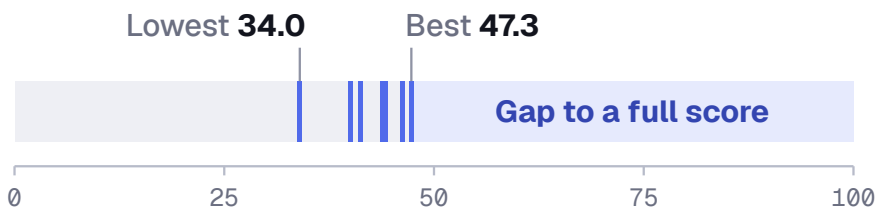
39.0%

of items needing a patch got
one with no critical defect
(best agent)

The seven agents finished within 13.3 points of each other

The top composite score was 47.3 out of 100. Qwen3.8 Max with OpenCode led the seven agents in the paper, and GPT-5.5 with Codex placed last at 34.0. Each paired a frontier model with a frontier coding harness.

FIGURE 2.1
The gap that no agent closed



Held-out split, 117 items. A score of 100 means every patch meets every requirement for its item.

Figure 2.2 splits each composite score into delivered quality and abstention recall, and adds each agent’s PO-clean delivery.

Critical defects are common

A critical defect is a failed PO criterion: an error or omission that would mislead readers or block the task. A patch is PO-clean when it has no critical defects. PO-clean delivery is the share of items that need a patch on which an agent delivered a PO-clean patch, so a missed patch counts against it.

FIGURE 2.2

The seven agents, measure by measure

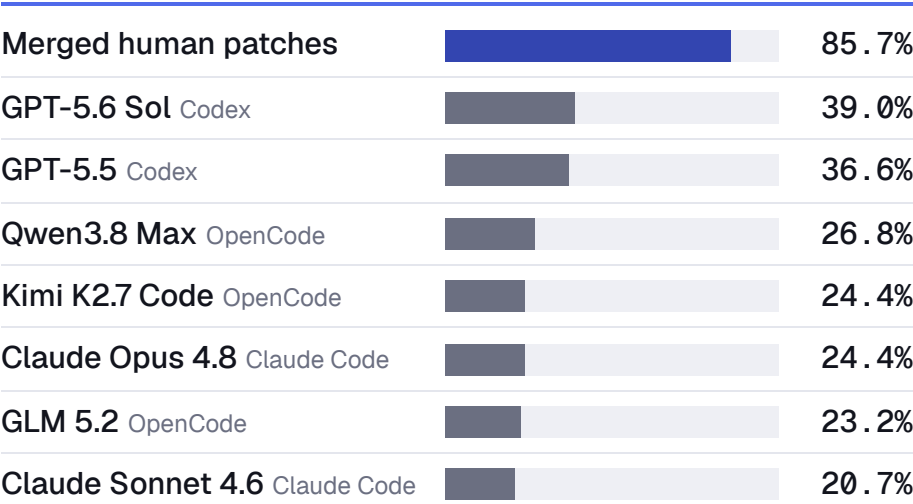
Agent	Composite score	Delivered quality	Abstention recall	P0-clean delivery
Qwen3.8 Max OpenCode	47.3	34.1	77.1%	26.8%
GPT-5.6 Sol Codex	46.2	44.1	48.6%	39.0%
GLM 5.2 OpenCode	44.2	30.1	82.9%	23.2%
Kimi K2.7 Code OpenCode	43.9	34.6	60.0%	24.4%
Claude Opus 4.8 Claude Code	41.2	27.1	85.7%	24.4%
Claude Sonnet 4.6 Claude Code	40.0	30.8	57.1%	20.7%
GPT-5.5 Codex	34.0	42.0	28.6%	36.6%

Held-out split.

We found that humans are better at judging what matters. Merged human patches were P0-clean 85.7% of the time, more than twice the best agent’s rate (Figure 2.3). We applied the same measure to the patches that maintainers merged. The best agent, GPT-5.6 Sol with Codex, delivered a P0-clean patch on 39.0% of the items, or 32 of 82.

FIGURE 2.3

Human patches were PO-clean more than twice as often



Agent rates are PO-clean delivery on the 82 held-out items that need a patch. The human rate is a share of merged human patches.

Human patches aren’t perfect either. The paper’s appendix lists four merged human patches that fail a PO criterion. One is a deploy command for Pants, the build tool in chapter 5, whose stray slash keeps it from resolving its target. Another is an OpenCost procedure that creates a Kubernetes Secret in one namespace and mounts it from another, so the procedure fails.

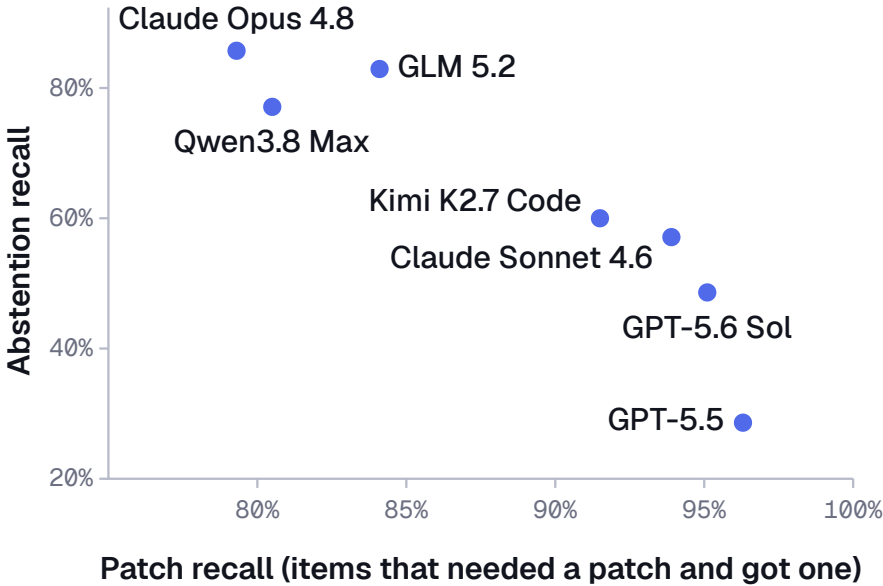
Agents make mistakes in both directions

No agent was strong at both deciding and writing. GPT-5.5 with Codex wrote a patch for 96.3% of the items that needed one, a rate called patch recall. But it left the docs alone on only 28.6% of no-change items. Claude Opus 4.8 with Claude Code

made the opposite trade: it left the docs alone on 85.7% of no-change items, but wrote a patch for only 79.3% of the items that needed one.

FIGURE 2.4

Agents trade one kind of mistake for another



Held-out split.

Treat small gaps between agents as ties

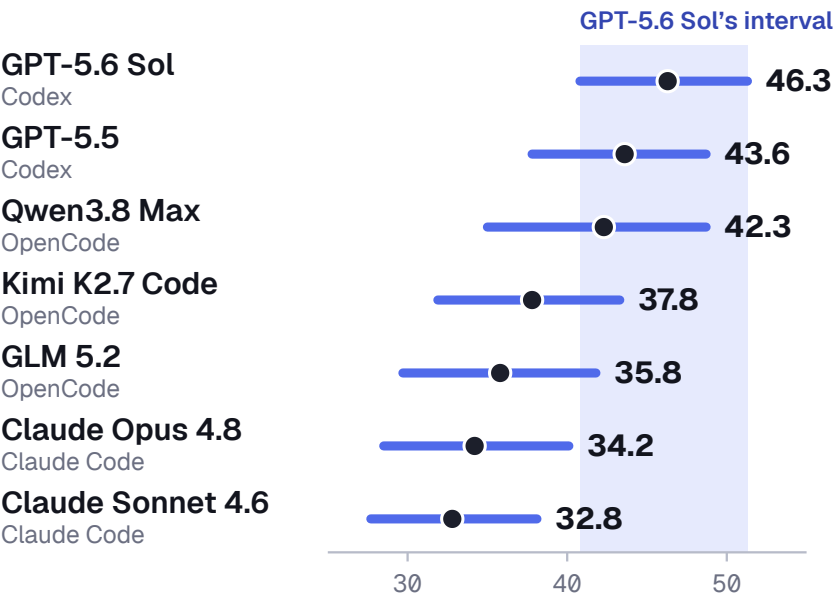
On conditional quality, GPT-5.6 Sol with Codex scored highest, at 46.3. Conditional quality averages only the patches an agent actually produced, so unlike delivered quality, it ignores missed patches. The leaderboard labels it Patch quality.

Each mean has a 95% confidence interval, a range that shows how precisely the data pins the mean down. The interval for GPT-5.6 Sol with Codex overlaps the intervals of the next four agents.

Each agent also ran each item once, so these results don't show how much a rerun would vary.

FIGURE 2.5

Most of the field overlaps



Conditional quality, with 95% intervals from resampling repositories.

Cloud agents, including Promptless, show the same weaknesses

We also ran three cloud agents on the same held-out split. A cloud agent is a commercial product that runs its own agent as a hosted service, with a model and harness its vendor chooses. They aren't part of the paper's ranking, so we report them as supplemental results. Two of the paper's authors work at Promptless, which builds documentation agents.

Promptless delivered a PO-clean patch on 40.2% of the items that needed one, so most of those items still didn't get a patch free of critical defects. It also wrote an unneeded patch on 37.1% of no-change items, 13 of 35. Expert review remains necessary.

FIGURE 2.6
Supplemental results for three cloud agents

System	Composite score	Delivered quality	Abstention recall	PO-clean delivery
Promptless Assessed September 2026	54.8	48.5	62.9%	40.2%
Devin Assessed August 2026	53.0	39.6	80.0%	35.4%
Mintlify Assessed August 2026	36.7	29.5	48.6%	24.4%

THE PAPER'S TOP AGENT, FOR REFERENCE

Qwen3.8 Max OpenCode	47.3	34.1	77.1%	26.8%
--------------------------------	------	------	-------	-------

- Same 117 held-out items as the paper's seven agents.
- We told the cloud agents not to use the internet. We verified that Promptless didn't browse, and we couldn't verify the same for the other two.
- Promptless states that it ran its production agent, with no changes made for DoGBench.

3



How docs agents fall short

Agent patches often sound right and still leave out what the reader needs to finish the task. We audited 1,267 submissions, each one agent's answer on one item. For each one, we read the agent's trajectory, the record of its steps, and traced each problem to its cause.

45.5%

of submissions had a task-completion gap, the most common failure mode

1.3%

of submissions had no material defect

Accurate text still leaves the reader stuck

FIGURE 3.1

Missing pieces outnumber wrong facts



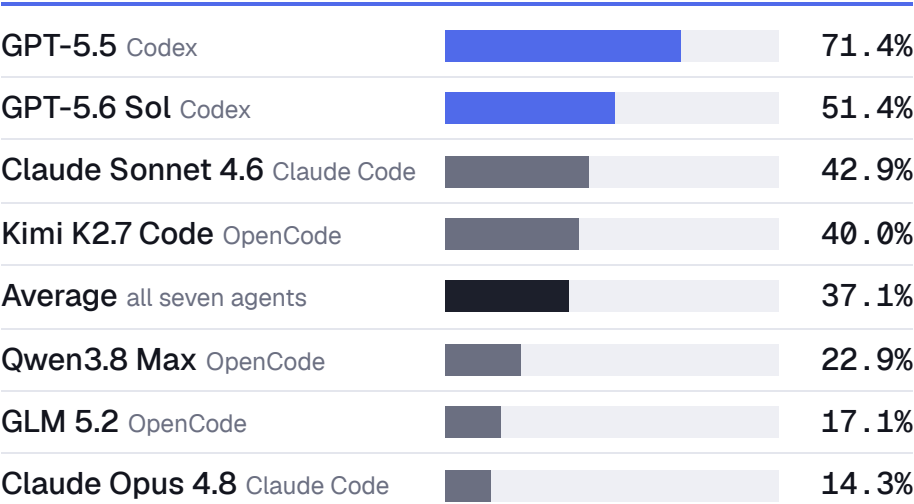
Audit of 1,267 submissions. Blue bars: defects found in at least 10% of submissions. Gray bars: less common defects.

The most common failure mode is the task-completion gap, and a patch can have one even when everything it says is accurate. In 45.5% of submissions, the patch left out a piece the reader needed, such as a decision point, a step, a check, or a recovery path. Fluent prose makes these easy to miss, because the patch reads well and most of it is true. Technical inaccuracy came second, at 36.6%. One patch can have several failure modes, so the rates overlap.

Behind these failure modes, agents have no model of how the product is actually used. That shows up in three patterns of behavior.

1. Agents don't know when not to write

FIGURE 3.2
Some agents write a patch for most changes that need none



Held-out split. The rate is 100 minus abstention recall.

Agents are eager to help, but sometimes that creates more work. On average, the seven agents wrote an unneeded patch (over-editing) for 37.1% of the 35 held-out no-change items. The rate ran from 14.3% for the most cautious agent to 71.4% for the most eager.

An unneeded patch usually starts with a misleading cue, like an internal name that sounds user-facing, or an existing page that mentions the component. Once an agent decides to edit, it looks for a place to put text and rarely asks again whether docs are needed.

Agents document three kinds of change that readers don't need:

- Internal refactors, renames, and regenerated types that don't change the public contract, the behavior users rely on
- Performance, test, continuous integration (CI), and dependency changes with no effect a user can see
- Changes to generated files and reference artifacts that people shouldn't edit by hand

Unneeded edits aren't harmless. Each one still has to be read and rejected. If your team merges one, it publishes implementation details that users can't act on, as if they were part of the product. Then the page has to be versioned, translated, and kept true, and the next agent run reads it as a pattern to follow.

2. Agents don't question the status quo

Agents take what the docs already say, and what they leave out, as the rule. If an old section breaks the style guide, the agent treats it as precedent and keeps breaking it. If a feature went undocumented by mistake, the agent assumes the omission was deliberate. When a related code change arrives, it skips the update instead of filling the gap.

The same habit causes missed patches (under-editing). A common trap is treating missing coverage as proof that a topic is out of scope, when the gap is the problem to fix. Agents also read a change's location in the code, or weak keyword overlap with the docs, as a sign that readers aren't affected. They dismiss real public interfaces in parsers and configuration modules as implementation details.

From the agent's perspective, deliberate exclusion and an unfilled gap produce the same search result. Documentation owners bring context that a code change alone may not reveal.

3. Agents jump to conclusions and focus too narrowly

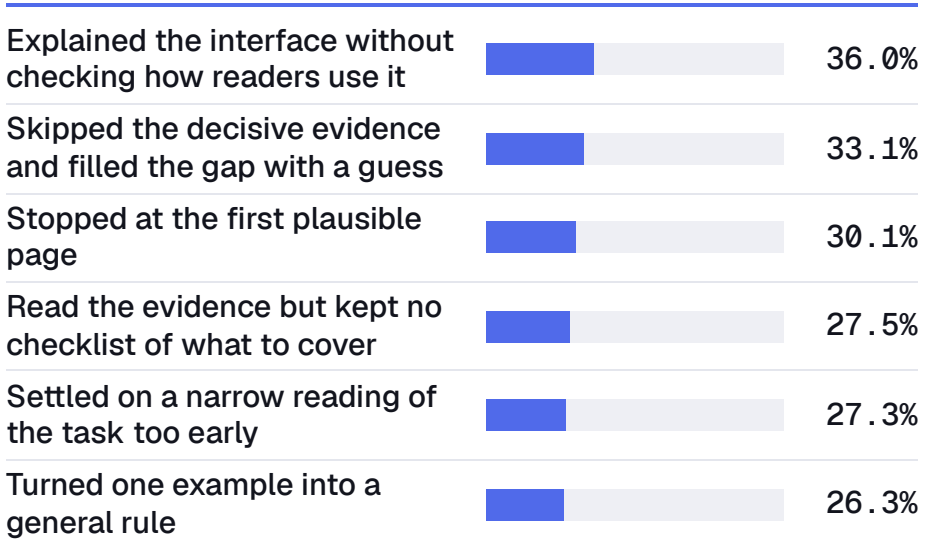
Agents switch from investigating to drafting too soon, because they have no reliable rule for when they know enough. This is premature closure, and it runs through many of the root causes we found. In 33.1% of submissions, the agent found related material, stopped short of the decisive evidence, and filled in with a plausible guess. Decisive evidence is the code change or contract that settles the question. In 30.1%, the agent stopped at the first plausible page and missed the other pages the same change affected. The most common root cause, at 36.0%, was explaining an interface without checking how readers use it.

It isn't about effort. The root causes showed no clear link to trajectory length, whether measured by turns or by token use. A token is a unit of text that a model reads and writes. Effort didn't rise with task difficulty either, which we defined by how the other

agents fared on an item. The correlations were 0.034 for tokens processed, 0.034 for trace events (the logged steps of a trajectory), and 0.030 for tool actions. Every confidence interval included zero. An agent can stop investigating early and still spend a lot of tokens drafting.

FIGURE 3.3

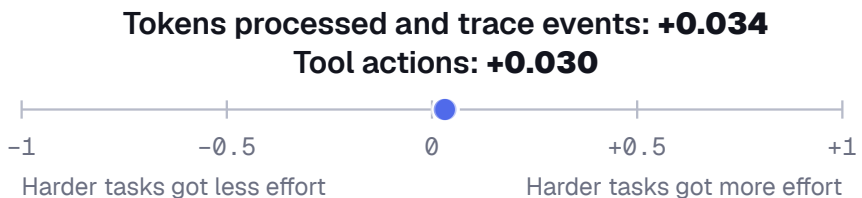
Six habits behind most failures



Trajectory analysis of 1,267 submissions. Several causes can apply to one submission.

FIGURE 3.4

Harder tasks did not get more effort



Rank correlations within each agent.

A stopping rule would tell an agent when it understands the request, the evidence, and the documentation impact well enough to start writing. SeekBench, a benchmark of search agents, finds the same problem. Agents trained with reinforcement learning answered before they had enough evidence in 76.5% of the trajectories that SeekBench evaluated.

Three likely causes didn’t explain the failures

We tested invention, training data, and project size.

Less invention, more misstated facts

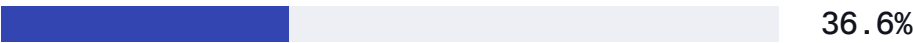
Frontier agents don’t invent features as much. More often, they take something true in one case and present it as true in every case. Fabricated content, such as a flag or an endpoint that doesn’t exist, showed up in only 6.1% of submissions, against 36.6% for technical inaccuracy. In those inaccuracies, the feature is real, but the patch overstates its scope, default, or compatibility. Claims like this are as unfounded as hallucinations, and harder to spot.

FIGURE 3.5

Distorted facts are six times as common as invented ones

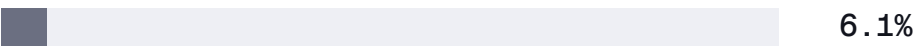
Technical inaccuracy

A real feature described wrongly



Fabricated content

A feature that does not exist



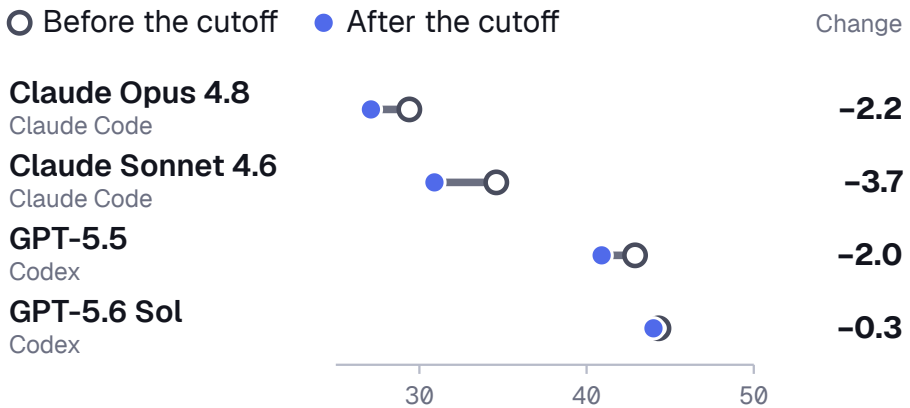
Audit of 1,267 submissions.

Training data and project size showed no clear effect

Four of the agents use a model with a published knowledge cutoff, the date its training data ends. Each of those agents scored lower on items dated after its model's cutoff, but every confidence interval included zero. Popular repositories gave agents no consistent advantage over less popular ones. Doubling the size of a project's docs changed scores by -0.98 points, and its confidence interval also included zero.

FIGURE 3.6

No clear effect from a model's knowledge cutoff



All 205 items that need a patch.

4



Five lessons for managing an agent

Agents can usually describe what a feature does. They're weaker at writing for a reader who came to decide something or finish a task. You bring context that a code change alone may not reveal, such as who the reader is, what they're trying to accomplish, and how the change affects their workflow.

45.5%

of submissions had a task-completion gap, the most common failure mode

33.1%

of submissions stopped short of the decisive evidence and guessed

DoGBench didn't test these lessons. We tried prompts, skills, and explicit verification informally, without a controlled comparison, so measure each lesson on your own work. The rates in this chapter come from our audit of 1,267 submissions.

Lesson 1: Help the agent understand your company, your docs, and your workflows

FIGURE 4.1
Three kinds of context an agent cannot infer

A	Conventions	Terminology The words you use and the ones you avoid
		Style guide How your docs should read
		Authoritative sources Where the truth lives
B	Audience	Personas Who reads your docs
		Goals What they came to do
		Common user journeys The paths they take through the product
C	Information architecture	Progressive disclosure What goes in an overview and what goes deeper
		Entry points Search, navigation, related guides, and examples
		Content graph Pages that link, mirror, or must change together

Write down what you'd otherwise explain in every review. Put it where the agent loads it every time, such as a style guide, an instructions file in the repository, or a skill. A skill is a file of reusable instructions that an agent loads when a request calls for it. A new writer picks up this context over time, but an agent starts every request without it. So a patch can be accurate in isolation and still fail inside the documentation system.

WHY IT MATTERS

20.7%

of submissions put content where readers were unlikely to look

13.3%

left a related page out of date

Lesson 2: Ask the agent to do research, and ask for a report with predefined fields

Require evidence before drafting. Give the agent the sources a writer would use. Then, before it drafts anything, ask for a short research report with predefined fields, such as the scope and impact of the change and any open questions. A report with blank or vague fields shows you what the agent hasn't verified, before any of it reaches a patch.

WHY IT MATTERS

33.1%

of submissions stopped short of the decisive evidence and guessed instead

FIGURE 4.2

Give the agent sources, and ask for a report

GIVE THE AGENT

Relevant sources Code changes, issues, specs, and support tickets

Connections to live systems Trackers, chat, and knowledge bases, through Model Context Protocol servers or similar tools

Skills Reusable instructions for your repository, tools, and review steps

ASK FOR THESE FIELDS

Scope and impact of the change The old behavior, the new behavior, and who the change affects

Verified boundaries Conditions, defaults, exceptions, and compatibility limits

Documentation impact map What the docs cover today, what they miss, and which other pages the change touches

Open questions and conflicts What is still uncertain, where sources disagree, and what to check before drafting

Lesson 3: Check that the patch includes the why and when, not just the what and how

Agents describe what a feature does, not the decision it supports. A reader usually comes to the docs to decide something or to reach a goal. A patch that only lists behavior leaves that reader to work out the rest alone.

WHY IT MATTERS

19.9%

of submissions lacked audience and purpose framing

26.5%

covered the main task but left out the rationale, limits, examples, or secondary cases

EXAMPLE FROM DOGBENCH

A Strawberry GraphQL patch correctly explained how to select an older Apollo Federation version, but not why a reader would want to, which is to upgrade Strawberry while staying compatible with an older Apollo Router or Gateway. It documented the setting and omitted the decision the setting exists to support.

Check that every patch answers three questions:

1. Who is this for?
2. When would they choose it, and why?
3. What must they get right?

Lesson 4: Walk through the reader's whole task

Agents don't understand that a change is part of a larger workflow. Readers need to complete the whole task, start to finish. Walk the reader's path through the patch: land, set up, decide, do, verify, and recover. Check that every step is there, in order, and that the patch says what to do when a step doesn't work.

WHY IT MATTERS

45.5%

of submissions had a task-completion gap, the most common failure mode

21.0%

left out prerequisites such as permissions, dependencies, credentials, versions, or settings

FIGURE 4.3

Walk the reader's path, stage by stage

STAGE 1 Land

Can the reader find the right page when they first notice the problem?

STAGE 2 Set up

Are permissions, credentials, versions, and settings covered?

STAGE 3 Decide

Can the reader tell which option fits their situation?

STAGE 4 Do

Is every step there, in order, with branches where needed?

STAGE 5 Verify

Can the reader tell that it worked?

STAGE 6 Recover

Does the reader know what to do when it did not work?

Lesson 5: Fact-check claims, especially their scope and applicability

Defaults, versions, lifecycle, and compatibility are where **accurate sentences go wrong**. Check each key claim for the conditions it depends on.

WHY IT MATTERS

36.6%

of submissions had a technical inaccuracy, often a claim stated more broadly than it holds

6.1%

contained fabricated content

EXAMPLE FROM DOGBENCH

An agent wrote that Helm 4 uses Server-Side Apply by default when installing or upgrading releases. That default applies only to new installations. Releases created with Helm 3 keep using client-side apply after an upgrade unless the user switches them. Claims like this take something true under narrow conditions and present it as true in general.

FIGURE 4.4

Five things to check in every key claim

Scope

Defaults

Versions

Compatibility

Exceptions

5



A complete patch changes two pages

We follow one item from the public development split, pantsbuild/pants #23219, from request to patch. Ten agents attempted it, the seven agents in the paper and the three cloud agents, and their scores ran from 25.0 to 92.3.

92.3

the best score on this item,
from GLM 5.2 with
OpenCode

25.0

the score of Kimi K2.7 Code
with OpenCode, whose
trajectory we trace step by
step

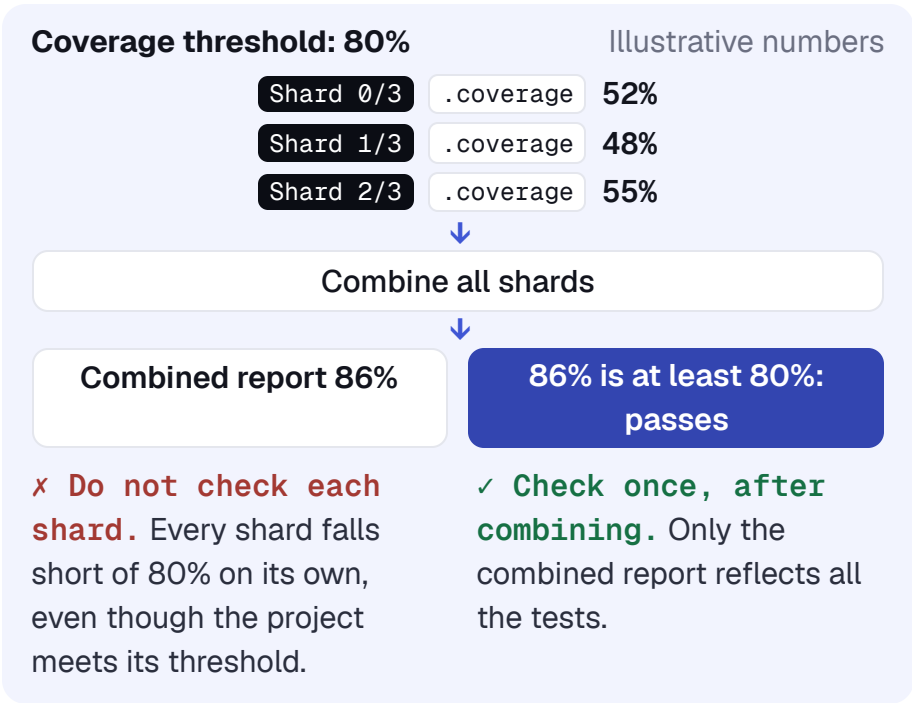
The request spans several CI jobs

Pants is a build tool that can split a project’s Python tests across several continuous integration (CI) jobs. Each job, called a shard, runs only some of the tests.

Test coverage is the share of the code that the tests exercise, and a shard’s coverage report covers only that shard’s tests. The team wants to know whether all the tests together meet the project’s coverage threshold, its pass mark. The request asked for docs that tell readers to save the raw coverage file from every shard and combine the files after the shard jobs finish. Readers then check the threshold on the combined report.

FIGURE 5.1

Check the threshold once, after combining



Illustrative numbers. The item comes from a documentation request, not from a code change.

The request sounds like “add a paragraph about coverage.”

Before drafting, an agent has to settle one fact: Pants combines coverage within a single run, but CI shards are separate runs. The procedure also has to reach readers who start on either of two pages.

The highest-scoring patch covers both pages

The highest-scoring patch adds the full procedure to the **Python test page**, and a caveat on the **sharding guide** links to it. A reader who starts on either page reaches the procedure.

FIGURE 5.2

Two pages, one procedure

docs/python/goals/test.mdx [Python test page · full procedure](#)

```
## Coverage
... fail_under and coverage configuration ...
+ ### Combining coverage from sharded test runs
+ 1. Request the raw report in every shard
+ 2. Save each shard's .coverage file
+ 3. Combine all files after the shard jobs finish
+ 4. Check the threshold on the combined total
## JUnit XML results
```

docs/using-pants/advanced-target-selection.mdx [Sharding guide · caveat and link](#)

```
## Sharding the input targets
... --test-shard=k/N explanation ...
+ Each shard checks the threshold against its own coverage only.
+ See Combining coverage from sharded test runs.
```

Paraphrased from the highest-scoring patch. Lines marked + are added by the patch.

A warning without a procedure scored 25.0

FIGURE 5.3

The agent's note states no risk and makes a required step sound optional

```
docs/python/goals/test.mdx Python test page · the agent's note, abridged

## Coverage
... fail_under and coverage configuration ...
+ :::note Coverage with --shard
+ ... each shard is an independent run over roughly 1/N of the targets ...
+ Coverage data and reports are therefore generated per shard, in separate sandboxes with distinct absolute paths.
...
+ You will also likely want [coverage-py].report to include "raw" ...
+ ... fail_under is evaluated against that shard's partial data ...
+ :::
```

- 1 **The risk is never stated.** The note gives the context but not the consequence: a threshold that fails in every shard. Criterion C2, P0.
- 2 **A required step sounds optional.** The words “likely want” make the raw report sound optional, but it produces the .coverage files that the combine step needs. Criterion C5, P0.

The agent's patch and rubric verdicts, abridged. Lines marked + are added by the patch.

Kimi K2.7 Code with OpenCode correctly warned that a shard's coverage is partial, and then stopped. Its one note leaves readers with no working procedure. Its patch failed four P0 criteria and one P1 criterion.

FIGURE 5.4

Neither page leads readers to a working procedure

docs/python/goals/test.mdx
Python test page · after the note

```
## Coverage
+ :::note Coverage with --shard ... :::
No steps to save each shard's .coverage file, combine the files, or check the total
## JUnit XML results
```

3

docs/using-pants/advanced-target-selection.mdx
Sharding guide · no change

```
## Sharding the input targets
... --test-shard=k/N explanation ...
No caveat and no link to the coverage procedure
```

4

- 3 There is no working procedure.** The note never tells readers to save, combine, and check the files, so the page lacks the complete workflow. Criteria C3 and C4, P0.
- 4 Readers of the sharding guide never find out.** The sharding guide gets no caveat and no link. Criterion C10, P1.

The agent's patch and rubric verdicts, abridged. Lines marked + are added by the patch.

Each failure traces back to a step in the trajectory

The agent never mapped how a CI user gets from partial shard results to one checked coverage total. Figures 5.5 and 5.6 trace each failure to the steps of the trajectory where it began, and name the lesson that could have caught it. Each step groups related actions, so the step numbers don't count turns.

FIGURE 5.5

Steps 1 to 4: the agent finds the pages late and misses what the code means

STEPS 1 TO 3 Finds the pages late

It searches .md files and at first misses the relevant .mdx docs. It finds the Python test page and the sharding guide, but never lists the pages that need changes.

CATCH IT · LESSONS 1 AND 2

Ask for a research report that names the reader's entry point, every affected page, and the source for each claim.

STEP 4 Finds the code, misses its meaning

It mentions the raw report but never checks what raw produces or when the threshold check runs.

CATCH IT · LESSONS 2 AND 5

Require the research report to state that raw produces the .coverage file. Check the scope of each step in the patch against the code that controls it.

FIGURE 5.6

Steps 5 to 10: the agent never plans the reader's task, so it ships a warning

STEPS 5 TO 8

**Reads pages
without planning
the task**

It rereads the Python test page and the sharding guide. Its plan stays “inspect docs, decide, edit.” It never maps how a CI user gets from partial shard results to one checked total.

CATCH IT · LESSONS 3 AND 4

Before the agent drafts, ask it to name the reader and their goal. Then ask it to list the sequence: request raw data, save every shard's file, combine the files, and check the threshold.

STEPS 9 AND 10

**Ships a warning
instead of
instructions**

It adds one note to the Python test page. The patch has no save or combine steps, no final threshold check, and no link from the sharding guide.

CATCH IT · LESSONS 1 AND 4

Walk the reader's route from the sharding guide to the procedure. Then try to finish the task using only the proposed text.

A better agent run understands the request first

GLM 5.2 with OpenCode scored 92.3 on the same item. It worked out what Pants does within one run, what separate CI jobs must do afterward, which facts come from the code, and

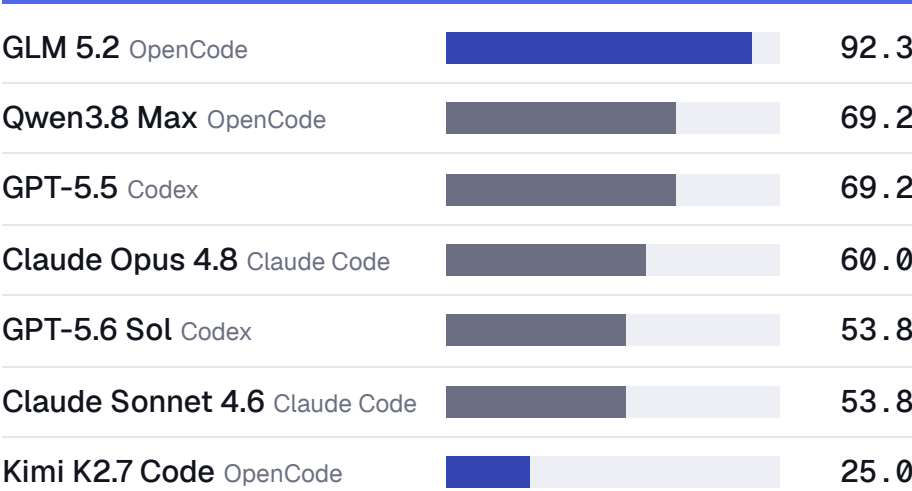
which page a reader reaches first. The two agents received differently worded versions of the request, so this isn't a controlled comparison.

FIGURE 5.7
An effective run, mapped to the five lessons

Stage	What the agent does
Map the docs Steps 1 to 2 Lesson 1	Reads the Python test page and the sharding guide.
Settle the facts Steps 3 to 4 Lessons 2 and 5	Reads coverage_py.py and test.py. Separates combining within one run from combining across CI jobs.
Define the reader's goal Step 5 Lesson 3	Names the goal, an accurate project-wide threshold from separately run shards.
Write both pages Steps 6 to 7 Lessons 1 and 4	Adds "Combining coverage from sharded test runs" to the Python test page, plus a caveat and link in the sharding guide.
Inspect the result Step 8 Lesson 5	Rereads the edited section and checks the link and wording.

A warning tells the reader that something can go wrong. A procedure tells them how to finish the job.

FIGURE 5.8
Same task, scores from 25.0 to 92.3



Scores for the seven agents in the paper. Blue bars mark the two agents this chapter follows.

6



Turn the findings into review habits

Review the decision first. Before reading the prose, ask whether the change needs user-facing docs at all. Then work through the reviewer's checklist, and use the questions for operations leads when your team decides how to adopt an agent.

85.7%

of merged human patches were PO-clean, more than twice the best agent's rate

175

public items you can run your own agent against

The reviewer's checklist

Work through the stages in order, because a failed early check can make the later stages unnecessary. For example, when the product change affects nothing a user can invoke, see, or act on, the patch is unneeded, and you can stop there.

STAGE 1 · REVIEW THE DECISION, NOT JUST THE PROSE

- ☐ Does the product change behind this patch affect anything a user can invoke, see, or act on?
- ☐ If the agent left the docs alone, is the missing coverage itself the gap to fix?
- ☐ Did the agent show its sources, and does each claim trace to one?

STAGE 2 · READ FOR THE READER'S WHOLE TASK

- ☐ Can a reader find the changed page from where they first notice the problem?
- ☐ Are the prerequisites stated, such as permissions, versions, and settings?
- ☐ Does the patch say who the feature is for, and when and why a reader would choose it?
- ☐ Using only the text, can the reader do the task, tell that it worked, and recover when it didn't?

STAGE 3 · CHECK SCOPE AND CONDITIONS, NOT JUST FACTS

- ☐ Is each default stated with the conditions it depends on?
- ☐ Does each claim hold for every version and setup a reader could apply it to?
- ☐ Did the agent check the request's own claims against the code?

STAGE 4 · LOOK BEYOND THE PAGE THAT WAS EDITED

- ☐ Do other pages describe the same behavior, and are they now out of date?
- ☐ Does every page a reader might start from link to the new content?
- ☐ Does the patch follow your style guide, even where old pages don't?

Questions for documentation and content operations leads

Who decides whether a change needs documentation?

Agents misjudge this in both directions. Keep the decision with a person, or review the decision before you review any prose.

Where do your conventions live?

Terminology, audience, and information architecture have to exist in writing before an agent can load them.

How will you evaluate an agent before you adopt it?

Run it on your own past changes, including changes that needed no docs, and review the results against your team's standards.

What will you measure after adoption?

Track how often reviewers reject unneeded patches and how many critical defects they catch. Both numbers show whether your context and review steps are working.

Who reviews, and is their time in the plan?

Budget for expert review from the start, because it catches the critical defects that agents miss.

Run DoGBench on your own agent

The code for running and scoring DoGBench is on [GitHub](#), and the 175-item public development split is on Hugging Face. Develop and debug on the development split. The leaderboard on [dogbench.ai](#) compares agents on the held-out split. For reproducibility, we're publishing 735 complete agent trajectories from the seven agents in the paper, so you can study each agent's steps.

Have results to share? Submit your agent's DoGBench results for review. We review each entry by hand and will follow up about verification and evaluation requirements.



Benchmark and scoring code

github.com/Promptless/dogbench



Development split on Hugging Face

[huggingface.co/datasets/promptless-research/
dogbench-dev](https://huggingface.co/datasets/promptless-research/dogbench-dev)



Leaderboard submission guide

[github.com/Promptless/dogbench/blob/main/docs/
leaderboard-submission.md](https://github.com/Promptless/dogbench/blob/main/docs/leaderboard-submission.md)

The paper, data, example items, and readout slides are at dogbench.ai. The slides are a short deck for presenting the findings to your team or leadership.

What DoGBench doesn't measure

DoGBench has these limits:

- **Each agent ran each item once.** The results do not show whether an agent would make the same decision, or write a patch of the same quality, on another attempt.
- **Agents had no internet access.** The seal keeps agents from finding the merged answer, but it also removes a tool that docs agents use in ordinary work. We reviewed 683 trajectories from items on which no agent produced a mergeable patch. Only 1.5% of them tried to fetch outside evidence that a correct patch needed. Internet access might therefore have helped on a small number of items.
- **The rubrics and scores can contain errors.** A scoring model applies the rubrics, and human validation covers only part of the evaluation.
- **One of us settled disagreements in the human reference.** The scoring model's 93.9% agreement rate is therefore not independent agreement between two people.
- **The scoring model checks commands and examples against the evidence but does not run them.** The benchmark has no automated tests that run code samples or check links.
- **A few no-change labels may be wrong.** Some rest only on no documentation update appearing within 90 days, so those items may have needed docs that nobody wrote.
- **Secondary criteria count equally.** During human calibration, reviewers prioritized the P1 to P3 criteria differently. The dataset keeps the priority labels, and the scoring code accepts other weights.
- **The scoring model may miss low-information prose.** A general prompt to the scoring model flagged 3.2% of submissions for coherent but low-information prose. A prompt

that listed specific patterns flagged 7.6%. AI is not very good at detecting AI slop.

- **DoGBench covers existing products only.** Every item starts from a product and its current docs. DoGBench does not cover writing a product's docs from scratch or redesigning an information architecture.

Glossary

Abstention: An agent's answer that the docs need no change for an item.

Abstention recall: The share of no-change items on which an agent correctly abstained.

Agent: A model paired with a harness, such as Claude Opus 4.8 with Claude Code.

Cloud agent: A commercial product that runs its own agent as a hosted service, with a model and harness its vendor chooses. This guide reports three as supplemental results.

Composite score: The headline DoGBench score, out of 100. It is the harmonic mean of delivered quality and abstention recall, so it stays low unless both are high.

Conditional quality: The average patch score on only the items that need a patch and got one. The leaderboard labels it Patch quality.

Confidence interval: A range around a measured value that shows how precisely the data pins it down. An interval that includes zero means the true effect could be zero.

Criterion: One check in a rubric, such as whether a patch states a default correctly.

Critical defect: A failed PO criterion, meaning an error or omission that would mislead readers or block the task.

Delivered quality: The average patch score across all items that need a patch. A missed patch scores zero.

Development split: The 175 public items, for developing and debugging an agent.

Failure mode: A category of defect in the audit, such as a task-completion gap or a technical inaccuracy.

Harness: The tool that runs a model and gives it files and commands, such as Claude Code, Codex, or OpenCode.

Held-out split: The 117 items used for the main results. Their rubrics are not public.

Item: One real docs task, built from one pull request or one reported documentation gap.

Missed patch: A patch that an item needs but the agent never writes. It scores zero. The paper calls this under-editing.

No-change item: An item whose correct answer is to leave the docs alone.

PO criterion: A criterion whose failure alone would block the patch in review. Failing it caps a patch score at 60.

PO-clean: Describes a patch that fails none of its PO criteria.

PO-clean delivery: The share of items that need a patch on which an agent delivered a PO-clean patch. A missed patch counts against it.

Patch: The set of edits to the docs that an agent proposes for one item. A patch can add, revise, move, or delete content.

Patch recall: The share of items that need a patch on which an agent wrote one.

Premature closure: The habit of switching from investigating to drafting before the evidence is complete.

Rubric: The list of criteria for one item, built from that item's change and docs.

Scoring model: The model that applies each rubric criterion to a patch. DoGBench uses GPT-5.6 Terra.

Secondary criteria: The P1 to P3 criteria in a rubric.

Submission: One agent's answer on one item. The audit reviewed 1,267 of them.

Task-completion gap: The most common failure mode, in which a patch leaves out a decision point, step, check, or recovery path the reader needs.

Trajectory: The record of an agent's steps on one item, from first read to final answer.

Unneeded patch: A patch written for an item that needs no change. The paper calls this over-editing.

Contributors

Authors of the DoGBench paper: Frances Liu (Promptless), Manny Silva (Promptless and Doc Detective), Paige Calvert (Helm), Ayu Adiati (Mautic), and Sarah Sanders (PostHog).

Community reviewers: Dachary Carey, Fabrizio Ferri-Benedetti, and CT Smith.

A big thank you to the maintainers who reviewed items and wrote criteria, and to the Write the Docs community.

How to cite DoGBench

Cite the paper when you use DoGBench or the numbers in this guide:

```
@article{liu2026dogbench,  
  title = {DoGBench: Can Agents Meet Expert Standards for  
User-Facing Documentation?},  
  author = {Liu, Frances and Silva, Manny and Calvert,  
Paige and Adiati, Ayu and Sanders, Sarah},  
  year  = {2026},  
  note  = {Manuscript},  
  url   = {https://dogbench.ai/paper}  
}
```

About Promptless

Promptless builds AI agents that keep product documentation up to date, and it publishes this guide. Two of the DoGBench paper's five authors work at Promptless. The paper's results cover seven agents that Promptless didn't build. Promptless's own agent appears only in the supplemental cloud-agent results in chapter 2 (page 18), with the date of each result and the places where it falls short.

To see how Promptless works with documentation teams, visit promptless.ai.

AI agents can write polished documentation, but they often miss what readers need to complete a task. We tested seven agents on 292 real docs tasks from open source projects and scored every patch against the criteria documentation maintainers actually apply in review. This field guide covers the failure patterns, five lessons for managing an agent, and a reviewer's checklist.



47.3

out of 100, the best score among the seven agents



45.5%

of agent patches left the reader unable to finish the task

6.1%

contained fabricated content. Distorted facts were far more common.



Read it online, free
dogbench.ai/field-guide/
The web edition, the PDF, the paper, and the data

from Prompt**less**

