Is Agent Code Less Maintainable Than Human Code?

Shaswat Patel* [1] **Betty Li Hou*** [1] **Arun Purohit** [1] **Kai Xu** [1] **Jane Pan** [1] **He He** [1] **Valerie Chen** [2]

Abstract

Maintainability is a core dimension of software engineering, shaping how code is written, reviewed, and developed over time. While coding agents have demonstrated strong performance on single-issue tasks, it remains unclear how maintainable their code is when future agents build on top of it, potentially leading to compounding downstream effects. We investigate how agent code compares to human code in these maintenance settings, presenting **CodeThread**, a framework to construct controlled experiments from repository-level coding benchmarks. Applying CodeThread to four frontier coding agents and four benchmarks, we find that agents are less effective at resolving tasks when building on agent code compared to human code, with task resolve rate drops of up to 13.1%. Regression analysis reveals that many traditional software engineering maintainability metrics do not explain this difference. Instead, the clearest signals are subtler behavioral differences in agent code, such as changes to input validation and error handling, along with differences in downstream code size and task difficulty. These findings highlight the need to evaluate these systems not only by immediate task resolution but also by code maintainability, and point to potential sources of downstream errors introduced by agent code.

1. Introduction

Coding agents are increasingly integrated into software engineering workflows, with promises of "completing weeks of work in days" (OpenAI, 2026). However, a patch that passes tests and is functionally correct may still introduce structural complexity, unnecessary verbosity, brittle abstractions, or confusing implementation choices that only lead to failures when the code is later extended or modified in downstream tasks (Figure 1).

Code maintainability has been studied across many software development contexts, and static metrics have emerged to provide quantitative evaluations of maintainability. In particular, structural and size-based metrics, such as cyclomatic complexity (Ebert et al., 2016), Halstead volume (Halstead, 1977), and cognitive complexity (Campbell, 2018), have been used widely as proxies for the maintenance cost on a project (Riaz et al., 2009; Ardito et al., 2020). The adoption of coding agents raises questions about whether code authored by agents is more or less maintainable for future agents building on top of it compared to human-authored code. Recent work has measured agent code maintainability based on these static metrics (Wang et al., 2025; Orlanski et al., 2026), but does not suggest how this code compares against human code. Static metrics can often additionally be confounded by task difficulty and codebase conventions.

To this end, we present **CodeThread**, a framework to conduct controlled experiments comparing how well agents can build on and maintain agent versus human code. CodeThread transforms single-step, repository-level coding tasks (e.g., instances from SWE-Bench (Jimenez et al., 2024)) into two-step tasks that comprise an initial implementation task followed by a dependent downstream task. It then creates comparable agent and human code on the initial task while holding the downstream task author and test-based evaluation fixed, allowing us to isolate how authorship of the initial task affects downstream agent performance.

We demonstrate CodeThread on four frontier models—Claude 4.5 Sonnet (Anthropic), GPT-5 (OpenAI), GLM 4.7 (Zeng et al., 2025), and MiniMax 2.5 (MiniMax)—and on four software engineering benchmarks spanning a range of programming languages and task types. We find that building on agent code leads to drops in downstream task resolve rates more often than building on human code, with gaps in resolve rate of up to 13.1%. Using code instances from CodeThread, we identify sources of differences in downstream task performance on agent and human code, performing a logistic regression analysis on cases where the two conditions produce different outcomes. We find that traditional static maintainability metrics generally do not explain these differences. Instead, the clearest predictor of

[1]New York University [2]Carnegie Mellon University. Correspondence to: {spp9399, betty.li.hou}@nyu.edu. Code and data available at: https://github.com/shaswatpatel123/CodeThread.

Presented at the 2026 ICML AIWILD Workshop.

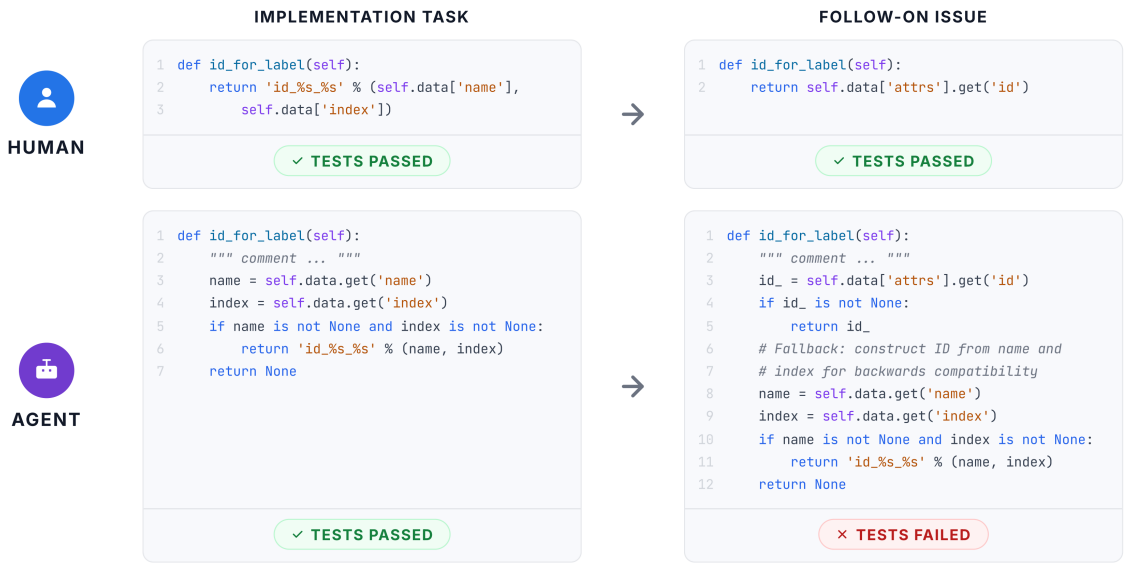

Figure 1. **Two pieces of code both being functionally correct does not mean they are equally maintainable.** An example instance where both the human and agent code pass the initial implementation task's tests, yet when an agent makes a subsequent change to both, the version built on human code passes while the version built on agent code fails.

cases where the downstream task fails on agent-authored code but succeeds on human-authored code is in subtle differences in input validation and error handling. Larger downstream edits and task difficulty also help characterize where the two conditions diverge.

Our contributions are as follows:

1. We introduce CodeThread, a framework for studying the maintainability of agent-authored code through two-step pull-request chains. CodeThread applies to any SWE benchmark, making it scalable across task types and languages, and provides controlled comparisons of agent to human code.

2. Applying CodeThread to multiple frontier models and benchmarks, we show that agent code is overall less maintainable than human code, leading to worse downstream task performance.

3. We analyze the sources of maintainability differences between agent and human code, showing that traditional proxies such as structural complexity and verbosity do not fully capture these differences, which instead often arise from subtler behavioral effects.

Together, CodeThread and our findings suggest that coding agents introduce maintainability costs that compound as their code is reused or extended in ways that current benchmarks and metrics fail to capture. We suggest directions for future work on the maintainability of agent code.

2. Related Work

Evaluating Agents on Sequential Tasks. Several recent benchmarks evaluate coding agents on sequences of dependent tasks rather than individual issues. CodeFlowBench decomposes Codeforces problems along their function-level dependency tree and requires the agent to implement each subfunction by reusing those it built in earlier turns (Wang et al., 2026). SWE-Bench-CL orders GitHub issues chronologically within each repository to evaluate whether experience from earlier issues transfers to later ones (Joshi et al., 2025). SlopCodeBench measures behavioral drift as agents repeatedly modify a codebase across long-horizon iterative trajectories (Orlanski et al., 2026). In contrast, our work grounds measurements in real GitHub pull requests rather than fully synthetic problems or hand-authored projects. Additionally, by varying the authorship of an intermediate patch while holding the downstream task fixed, we isolate the maintainability cost of agent-written code from the inherent difficulty of sequential development.

Maintainability of LLM-Generated Code. Large-scale GitHub analyses report rising code churn and duplication coinciding with AI-assistant adoption (Harding & Kloster, 2025; Cynthia et al., 2026; Huang et al., 2026; Liu et al., 2026), static analyses find that LLM-generated code exhibits more code smells than human-written code (Paul et al., 2025), and recent work shows that non-functional quality and design-constraint compliance degrade even when agent-generated patches pass tests (Sun et al., 2026; Yu et al., 2026). Prior work has also measured code-quality degradation in sequential tasks. MaintainCoder applies synthetic

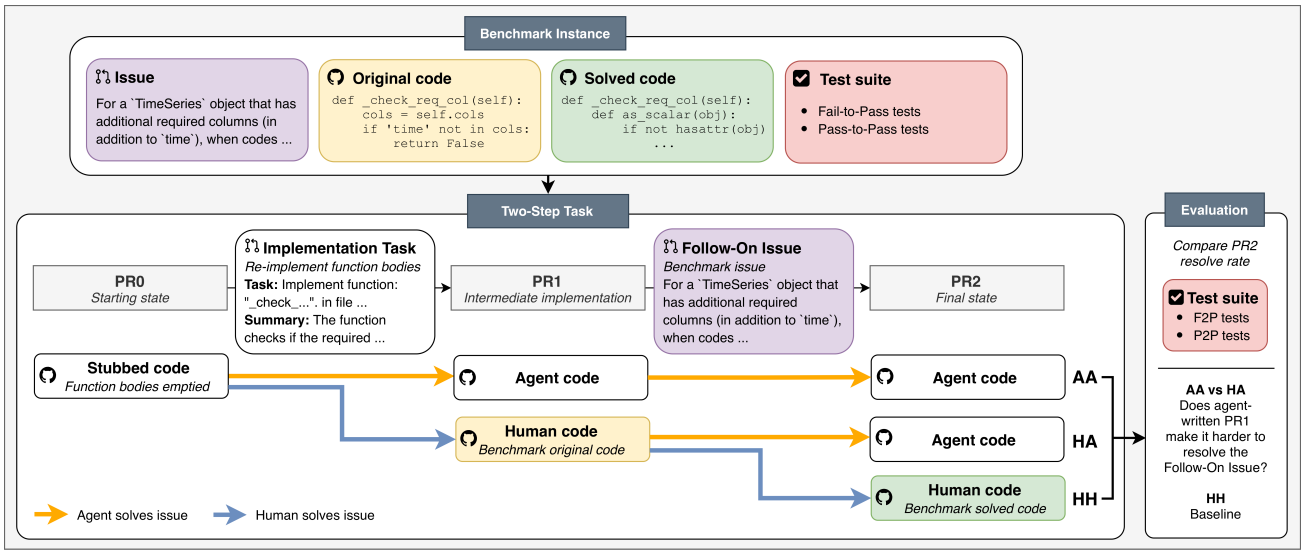

Figure 2. **CodeThread framework.** From the original benchmark instance, we construct a two-step task—an Implementation Task followed by a Follow-On Issue—producing three code states (PR_0, PR_1, and PR_2) and three conditions: AA (agent performs both steps), HA (human performs the Implementation Task, agent performs the Follow-On Issue on human code), and HH (human performs both steps). Comparing AA and HA isolates the effect of agent authorship on maintainability, holding the follow-on author fixed.

requirement changes to competitive-coding tasks and measures post-modification correctness alongside static metrics such as Cyclomatic Complexity and AST similarity (Wang et al., 2025). SWE-CI uses test pass rates over iterative edit trajectories as a maintainability proxy, treating later-stage pass rate as evidence of how well early decisions support subsequent maintenance (Chen et al., 2026).

Our study instead measures differs maintainability directly through downstream issue-resolution performance. Rather than relying primarily on static proxies or synthetic requirement changes, we ask whether an agent can successfully complete a dependent follow-on task when the intermediate code it inherits is written by an agent versus a human, creating controlled experiments to measure the effect of authorship. The resulting code can also be analyzed using static metrics and failure-mode annotations (Section 5).

3. CodeThread

We introduce CodeThread, a framework for measuring the downstream effects of building on agent code in comparison to human code (Figure 2). CodeThread transforms standard single-issue software engineering benchmark instances into dependent two-step pull-request chains (Section 3.1), allowing for controlled comparisons of downstream outcomes across authorship conditions (Section 3.2) with evaluation built in (Section 3.3).

CodeThread starts from an existing issue-resolution coding benchmark where instances consist of a written issue, a base commit, a human-authored ground-truth patch, and a test suite to determine whether the patch resolved the task. Example benchmarks that follow this format include SWE-Bench Verified (Chowdhury et al., 2024), SWE-Bench Multilingual (Khandpur, 2025), SWE-Bench Pro (Deng et al., 2025), and FeatBench (Chen et al., 2025).

3.1. Step 1: Create a Two-Step Task

From each benchmark instance, we construct a two-step chain consisting of two dependent tasks and three code states: the **Implementation Task** and **Follow-On Issue**, and code states $\mathbf{PR}_0$, $\mathbf{PR}_1$, and $\mathbf{PR}_2$. To create the first state, PR_0, we remove the target function bodies while preserving their signatures, leaving a skeleton version of the benchmark's initial code state. This provides a starting point from which different authors can implement the intended functionality, similar to the use of skeleton code in prototyping, where an initial code structure is later fleshed out with implementation details (Ryoo et al., 2008; Queirós, 2013).

The Implementation Task ($PR_0 \rightarrow PR_1$) produces human and agent implementations of the same task. Using an LLM, we create an issue asking the author to restore the missing functionality in the skeletonized code. We filter out PR_1 solutions that have already solved the benchmark issue by requiring them to pass all Pass-to-Pass tests, confirming that the original functionality is restored, while still failing all Fail-to-Pass tests, confirming that the downstream issue is not yet resolved. This ensures that PR_1 reflects the benchmark's initial code state rather than a partial or complete solution to the Follow-On Issue. Although models sometimes incidentally satisfy Fail-to-Pass tests while restoring

functionality, the filter retains a large majority of instances. Appendix A shows the prompt used to generate Implementation Task statements, as well as the prompt given to models when solving it.

The Follow-On Issue ($PR_1 \rightarrow PR_2$) is the original benchmark issue applied on top of the PR1 implementation. The problem statement and testing criteria for the Follow-On Issue are unchanged from the original benchmark, making it a downstream task whose difficulty may depend on the maintainability of the PR1 implementation. We evaluate whether Follow-On Issue is resolved using the original benchmark test suite: a PR_2 solution is considered correct if it passes both the Pass-to-Pass tests, confirming that existing functionality is preserved, and the Fail-to-Pass tests, confirming that the original benchmark issue has been fixed.

3.2. Step 2: Set-up Authorship Scenarios

Using the two-step issue resolution pipeline, we vary authorship on PR_1 and PR_2, creating a total of three comparable authorship scenarios: human PR_1 followed by human PR_2 (**HH**), human PR_1 followed by agent PR_2 (**HA**), and agent PR_1 followed by agent PR_2 (**AA**). The HH condition consists of the original developer's PRs provided by each benchmark, acting as the human baseline against which the other conditions are compared. Comparing AA to HA isolates the downstream effect of PR_1 authorship on PR_2 success, allowing us to directly evaluate whether agent code imposes a maintainability burden on subsequent agent work in comparison to human code.

A growing fraction of developers use AI agents to extend existing codebases (Li et al., 2025), which may contain originally human-written code or earlier agent-written code. Thus, we focus on the HA versus AA comparison. Both settings fix the agent as the PR_2 author and vary only the authorship of PR_1, isolating the effect of the prior code on the agent's downstream fix. The AH setting of a human performing the follow-on task on top of agent code would offer a complementary human-centric view (AH versus AA), but it requires recruiting a substantial number of developers and is out of scope for this study. By keeping the PR_2 author fixed, CodeThread is scalable and can be applied to many existing issue-resolution benchmarks without additional human studies.

3.3. Step 3: Evaluate agent vs human code

We assess the final code outputs, PR_2, from HA and AA in terms of resolve rate, which refers to whether the code, after both editing steps, successfully resolves the original benchmark issue. Differences in resolve rate indicate whether the authorship of PR_1 affects downstream issue-resolution success, providing a task-based measure of maintainability. The resulting code samples can also be compared along other dimensions of code quality since the task setup is controlled and the downstream issue is held fixed across authorship conditions. We show this in Section 5.

4. CodeThread on SWE Problems

4.1. Experimental Set-up

Task Types. We source a total of 1,377 instances from SWE-Bench Verified (Chowdhury et al., 2024), SWE-Bench Multilingual (Khandpur, 2025), SWE-Bench Pro (Deng et al., 2025), and FeatBench (Chen et al., 2025). The four benchmarks cover a range of software engineering tasks: bug fixing (BF), feature implementation (FI), and refactoring (RF), as shown in Table 1. Following Xu et al. (2025), we treat SWE-Bench Verified and SWE-Bench Multilingual as bug fix benchmarks. For SWE-Bench Pro, we use its native labels. We use FeatBench as a feature-implementation benchmark.

Models and Agent Configuration. In order to cover a breadth of model types, we evaluate four frontier models: Claude 4.5 Sonnet, GPT-5, GLM 4.7 and MiniMax M2.5, two proprietary frontier models and two publicly available open-weight models. We use SWE-Agent (Yang et al., 2024) as the scaffolding framework for all models. We follow the standard evaluation protocol of the underlying benchmarks and run each pair (model, authorship condition) once per instance in order to maintain tractable computational costs while providing coverage across all benchmark categories. We provide more details in Appendix C.

4.2. Results

Overall findings. Table 2 shows the PR_2 resolve rates under HA and AA conditions. The number of instances that pass the PR_1 filtering differs by model, so we compare models using the subset of instances shared across all models within each benchmark. Across these instances, we find that agents more often perform worse when building on agent code than on human code: **AA generally underperforms HA, with drops in downstream resolve rate of up to 13.1%.** Full scores for each model and benchmark are provided in Appendix B; we observe similar pattern in both the shared-instance analysis and the full per-model benchmark results. The largest drops occur for GLM 4.7 on SWE-Bench Pro, which contains longer, multi-file edits drawn from professional codebases, and with GPT-5 on FeatBench, a set of feature implementation tasks. We also observe four cases where AA matches or exceeds HA performance. We trace the source of this gap in Section 5.

Per-model patterns. Table 2 shows the HA-vs-AA gap across models. GLM shows the largest and most consistent effect of PR_1 authorship, with AA underperforming HA

Benchmark	# Instances		Filtered Subset		PR_1 (Avg.)		PR_2 (Avg.)		Task Types
	Total	Filtered.	#Repos	#Lang	Files	Funcs	Files	Funcs	
SWE-Bench Verified	500	441	12	1	1.12	1.60	1.18	1.71	BF
SWE-Bench Multilingual	300	171	34	7	1.29	1.90	1.84	3.28	BF
SWE-Bench Pro	731	641	11	4	2.43	5.51	5.22	10.70	BF, FI & RF
FeatBench	156	124	22	1	1.52	2.54	2.35	4.03	FI
Total	**1,687**	**1,377**	78	10	1.79	3.54	3.25	6.30	

Table 1. **Benchmarks used in our CodeThread evaluation.** We list each benchmark's full and filtered instance counts, retaining only instances requiring function-level edits. For the filtered subset, we report repository and language coverage (#Repos, #Lang), the mean files and functions modified per PR stage (PR_1, PR_2), and a breakdown of task types: bug fixes (BF), feature implementation (FI), and refactoring (RF).

on all benchmarks and dropping by 13.1% on SWE-Bench Pro. Claude and MiniMax also drop by around 3-8% across SWE-Bench variants, though FeatBench shows more mixed results. GPT-5 has the smallest gaps on SWE-Bench variants that may also partly reflect its lower HA baseline rather than greater robustness to agent code. GPT-5 drops by 12.5% on FeatBench, however, suggesting that maintainability costs may be more visible on feature implementation tasks.

Stratifying by task type. Table 3 shows that refactoring tasks have the largest average drop across models of 8.21%. Feature implementation and combined FI+RF tasks also show notable average drops of 6.25 and 6.82%, respectively. Bug-fix tasks show the smallest average drop, with an average 4.05%. These observations are consistent with prior work, which shows that refactoring and multi-file edits are harder for coding agents than localized fixes (Xu et al., 2025; Gautam et al., 2025).

5. Why does AA underperform HA?

We observe that agents building on agent code generally perform worse than on human code. To isolate the drivers of that gap, we focus on the discordant cohort, in which HA and AA produce different outcomes on the same task. Across all four models and benchmarks, there are 454 discordant instances, where HA wins 64.3% of the time and AA wins only 35.7%. After dropping instances with incomplete features, our analysis set is 405 instances (HA 64.7%, AA 35.3%). For each of these instances, we first extract a set of features from PR_1 and PR_2, detailed in Section 5.1, and fit a logistic regression model on these features to identify which push the outcome towards HA or AA.

5.1. Features

We use four feature categories to identify the root causes of the gap: static maintainability metrics to test traditional code-quality signals, patch localization features to compare files and functions edited under HA versus AA, PR_1 behavioral drift labels to capture differences in agent PR_1 missed by static metrics, and an instance difficulty score to control for task-level variation. We describe each below and provide implementation details in Appendix D.

PR_1 and PR_2 static metrics. Our first feature set comprises four static proxies for maintenance cost based on software engineering literature, previously discussed in Section 2 (Riaz et al., 2009; Ardito et al., 2020), spanning two dimensions of code quality. The first dimension is *structural complexity*, where we use Cyclomatic Complexity (CC) (Ebert et al., 2016), which counts linearly independent paths through a function, and Cognitive Complexity (CogC) (Campbell, 2018), which penalizes nested control structures. The second dimension is *verbosity*, where we use Halstead Volume (HV) (Halstead, 1977), based on operator and operand counts, and Logical Lines of Code (LLOC), which counts executable statements. For each metric, we compute two features per discordant instance: the difference between AA and HA at PR_1 and PR_2.

PR_2 patch localization features. Our second feature set measures files and functions that the agent edits. Prior work has shown that localization is one of the leading failure modes for coding agents (He & Roy, 2026; Liu et al., 2025). We focus on PR_2 because the PR_1 prompt explicitly lists the files and functions each chain must edit. The PR_2 problem statement does not, and the agent can freely choose which files and functions to edit. We measure this with two Jaccard overlaps: one over the set of edited files, and one over the set of edited functions. Higher values indicate that the HA and AA patches modify more similar parts of the codebase.

PR_1 behavioral drift labels. The third feature set captures differences between agent and human PR_1s that are not reflected in aggregate static metrics. Based on manual inspection, we observed that differences in exception types or swapped defaults can silently creep into the codebase. The agent might add an extra input-gating check that raises `ValueError` or uses `getattr` with `None` as the default, while the human's PR_1 would have raised an error. In both cases, neither set of prior features would distin-

	Resolve Rate % (↑)											
	Claude			GPT			GLM			MiniMax		
Benchmark	HA	AA	Δ	HA	AA	Δ	HA	AA	Δ	HA	AA	Δ
SWE-Bench Verified	**71.7**	66.7	−5.1	**62.6**	61.6	−1.0	**65.7**	62.6	−3.0	**74.7**	68.7	−6.1
SWE-Bench Multilingual	**53.1**	50.0	−3.1	46.9	**50.0**	+3.1	**62.5**	59.4	−3.1	**65.6**	62.5	−3.1
SWE-Bench Pro	**46.3**	38.6	−7.7	39.0	**39.8**	+0.8	**45.2**	32.0	−13.1	**43.6**	36.7	−6.9
FeatBench	50.0	**56.2**	+6.2	**56.2**	43.8	−12.5	**62.5**	56.2	−6.2	56.2	56.2	0.0

Table 2. **Comparing HA and AA PR_2 resolve rate.** We observe a consistent lower resolution rate under the AA condition than HA across nearly all model–benchmark pairs. Bolded values indicate the higher resolve rate within each HA/AA pair.

		Resolve Rate % (↑)												
		Claude			GPT			GLM			MiniMax			
Task	**N**	HA	AA	Δ	HA	AA	Δ	HA	AA	Δ	HA	AA	Δ	**Avg. Δ**
BF	210	**59.05**	51.43	−7.62	49.52	**50.00**	+0.48	**56.19**	52.38	−3.81	**61.43**	56.19	−5.24	−4.05
FI	104	**44.23**	41.35	−2.88	**41.35**	38.46	−2.88	**46.15**	30.77	−15.38	**42.31**	38.46	−3.85	−6.25
RF	70	**57.14**	50.00	−7.14	48.57	**51.43**	+2.86	**55.71**	40.00	−15.71	**54.29**	41.43	−12.86	−8.21
FI + RF	22	**27.27**	22.73	−4.55	27.27	27.27	0.00	**31.82**	13.64	−18.18	**27.27**	22.73	−4.55	−6.82

Table 3. **Task-type stratified resolved rate results by model on overlapping instances.** The dataset is stratified by task type into bug fixes (BF), feature implementation (FI), refactoring (RF), and feature implementation & refactoring (FI + RF). PR_2 resolve rates under HA and AA; Δ denotes the percentage-point change from HA to AA within each task type and model. Avg. Δ denotes the average % change across models for each task type.

guish the agent from the human. We use LLM-as-a-judge to detect this *input/error contract* (IEC) behavior, which is set to 1 when the agent's PR_1 alters input-validation or error-handling behavior.

Instance difficulty control. We add a difficulty control to ensure that HA is not winning simply because it is evaluated on easier tasks. We use a leave-one-out (LOO) instance difficulty score, defined as the mean resolution rate of the other three models on the same instance.

5.2. Modeling

Given these twelve features, together with benchmark and model fixed effects, we fit a logistic regression (LR) to identify which features push the outcome toward HA or AA. Let $Y_i = 1$ when HA resolves and AA fails on instance i and $Y_i = 0$ otherwise. The model is defined as:

$$\begin{aligned}\text{logit}\ \Pr(Y_i = 1 \mid X_i) = \beta_0 &+ \beta_{\text{IEC}}\,\text{IEC}_i + \boldsymbol{\beta}_J^\top \mathbf{J}_i \\ &+ \boldsymbol{\beta}_1^\top \boldsymbol{\Delta}_i^{\text{PR}_1} + \boldsymbol{\beta}_2^\top \boldsymbol{\Delta}_i^{\text{PR}_2} \\ &+ \boldsymbol{\beta}_D^\top \mathbf{D}_i + \beta_{\text{LOO}}\,\text{LOO}_i\end{aligned}$$

Static maintainability metrics are denoted with $\boldsymbol{\Delta}_i^{\text{PR}_1}$ and $\boldsymbol{\Delta}_i^{\text{PR}_2}$ for each of four AA−HA metric deltas (CC, CogC, HV, LLOC) at PR_1 and PR_2. For fault localization, the HA-vs-AA file and function Jaccards on the PR_2 patches are denoted by $\mathbf{J}_i = (J_i^{\text{file}}, J_i^{\text{func}})$. PR_1 behavioral drift is denoted by $\text{IEC}_i \in \{0, 1\}$, an input/error-contract indicator on the agent's PR_1. LOO_i denotes the leave-one-out instance difficulty score, and $\mathbf{D}_i$ is a fixed-effect for benchmark and model. We interpret coefficients as Odds Ratios (ORs), each of which captures how a feature shifts the outcome: a one-unit increase in feature k multiplies the odds of HA winning by OR_k. Coefficients with OR > 1 therefore tilt the outcome toward HA, and OR < 1 toward AA. In Figure 3, we similarly observe the effects of each feature on the direction of discordance by visualizing the distribution of each feature across HA-resolved and AA-resolved instances.

Feature	**OR (95% CI)**	***p***
$\Delta\text{CC}_{\text{PR}_1}$	0.92 (0.53, 1.61)	0.779
$\Delta\text{CogC}_{\text{PR}_1}$	1.00 (0.67, 1.48)	0.981
$\Delta\text{HV}_{\text{PR}_1}$	1.03 (0.69, 1.53)	0.885
$\Delta\text{LLOC}_{\text{PR}_1}$	0.92 (0.53, 1.61)	0.780
$\Delta\text{CC}_{\text{PR}_2}$	0.71 (0.38, 1.31)	0.271
$\Delta\text{CogC}_{\text{PR}_2}$	0.97 (0.69, 1.37)	0.866
$\Delta\text{HV}_{\text{PR}_2}$	0.93 (0.69, 1.25)	0.615
$\Delta\text{LLOC}_{\text{PR}_2}$	**1.88** (1.02, 3.46)	**0.042**
File Jaccard	0.77 (0.58, 1.01)	0.061
Func. Jaccard	1.22 (0.93, 1.60)	0.144
IEC	**1.83** (1.15, 2.92)	**0.011**
Inst. resolve rate	**1.42** (1.12, 1.78)	**0.003**

Table 4. **Effect of PR_1 and PR_2 features on HA and AA agreement.** The three significant predictors of whether HA or AA will resolve in the discordant pair are: $\Delta\text{LLOC}_{\text{PR}_2}$, IEC, and instance difficulty. All three tilt the outcome toward HA: IEC drift in agent PR_1, AA over-editing at PR_2 relative to HA, and easier instances each raise the odds that only HA resolves.

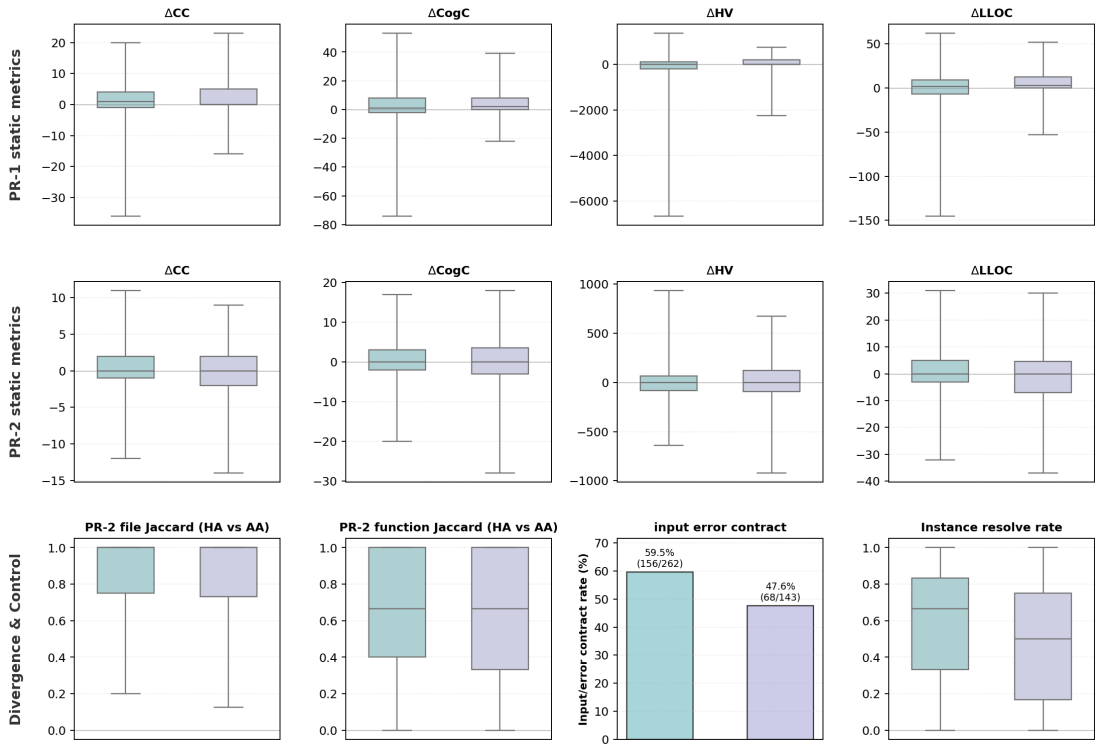

Figure 3. **HA vs AA-only wins are differentiated by behavioral drift, instance difficulty, and downstream code-size change.** Panels compare instances resolved only in HA (teal) versus in AA (lavender); whiskers show the 5th/95th percentiles. Most static maintainability metrics have similar distributions, except ΔLLOC at PR_2; input error contract and instance resolve rate also separate the two groups.

5.3. Findings

Which features predict HA and AA disagreement? Table 4 reports the fitted logistic regression on the discordant instances. Three features emerge as statistically significant predictors of the discordance direction, with all three tilting the outcome toward HA resolved while AA unresolved: IEC, ΔLLOC_{PR_2}, and Instance resolve rate. When agent PR_1 alters input-validation or error-handling behavior, the odds of HA winning are $1.83\times$ that of AA, and when AA's PR_2 adds more lines than HA's, HA is $1.88\times$ more likely to win over AA. The difficulty control feature signifies that instances where HA wins tend to be easier. The model recovers a modest but reliable signal (McFadden $R^2 = 0.069$, LR $\chi^2(18) = 36.1$, $p = 0.007$). Figure 3 corroborates these results as static maintainability metrics show near-identical distributions across HA-wins and AA-wins, while IEC drift, ΔLLOC_{PR_2}, and instance resolve rate show the clearest visual separation between the two groups.

Does PR_1 drift cause the unresolved outcome in PR_2? The IEC coefficient in Table 4 establishes the population-level signal, but population-level evidence does not tell us whether the drift actually drove the failure on a given instance. We focus on IEC for per-instance attribution because it is the only significant predictor in our model that identifies a specific, code-level mechanism. ΔLLOC_{PR_2} is a summary statistic and instance difficulty is a control variable, neither of which corresponds to a verifiable property of the agent's PR_1 that we can inspect per instance.

Two scenarios could decouple IEC drift from the downstream failure. First, the agent's PR_2 may rewrite the function containing the PR_1 input/error-contract divergence, removing the drift before evaluation. Second, the drift may persist into PR_2, but the failing test may exercise a different part of the code. We use an LLM-as-a-judge to attribute whether the failing test can be traced to the IEC drift. The LLM-judge sees the PR_1 behavioral drift label with its generated rationale, the AA PR_2 patch, and the AA test outcome. It answers whether the divergence persisted into PR_2, and whether the failing test can be trace back the divergence.

Table A4 reports the breakdown. The first scenario is rare: PR_2 rewrites the divergent function on only 6.9% of instances; the drift is unchanged in 85.9% of instances. On 20.6% of instances, the drift both persists and directly causes the failure, a signal the static metrics in Table 4 do not capture. The remaining 74.4% fail for reasons we have not isolated, and we leave this analysis to future work. The high persistence rate also suggests that on long-horizon tasks (i.e., chains longer than our two-PR setup), this drift

could compound across PRs and degrade downstream resolve rates further. Our setup cannot test this directly, and future work should assess the effects of these drifts over long-horizon tasks. The full prompt and output schema are in Appendix D.4.

6. Discussion & Limitations

We present limitations of our study. First, we consider only two-step pull request chains ($PR_1 \rightarrow PR_2$). Although this setup is sufficient to study whether the quality of an intermediate patch affects a downstream modification, it does not capture longer-horizon development trajectories, where technical debt may compound over multiple successive edits. Future work should explore constructing longer PR chains with function-level overlap across successive tasks.

Second, we filter PR_1 by instances where the agent PR_1 did not already address the Follow-On Issue. This filtering is necessary for the two-step chain to be well-defined, maintaining the separation between the two tasks so that the effect of authorship can be evaluated in isolation from functionality. This may introduce a selection bias, where restricting the analyzed instances to where the agent did not solve the Follow-On Issue at PR_1 does not represent the agent's natural behavior across the benchmark. The filter retains a large majority of instances, however, and we verify that cases where the model satisfies Fail-to-Pass tests in the Implementation Task are often incidental.

Third, our work does not study the effect of human-authored PR_2 on top of agent-authored PR_1 (AH) setting. Doing so would allow for a comparison of how difficult it is for human developers to build on agent code versus human code. Constructing this condition at scale would require recruiting developers to author PR_2 across 1,409 instances, which is infeasible in our current setting and difficult to control due to factors such as familiarity with the codebase. Future work could examine the AH setting on a smaller curated subset.

7. Conclusion

Across four benchmarks and four frontier models, building on agent code more often lowers downstream resolve rates than building on human code, with effects varying by model and task type. These differences are not well explained by static maintainability metrics alone; the clearest code-level signal is subtle behavioral drift in agent code, while larger downstream edits and task difficulty also indicate where the conditions diverge. Thus, building on agent code often introduces maintainability costs, which appear even on two-step chains and are likely to compound over many edits of a software project. As agent contributions become a larger share of working codebases, evaluation must shift from isolated task completion toward long-term maintainability.

Acknowledgments

We thank Carlos Jimenez, Nicholas Lourie, Varun Yerram, Shashwat Singh, Rico Angell, and Yueh-Han Chen for insightful feedback and discussion. This work was supported in part through the NYU IT High Performance Computing resources, services, and staff expertise.

Impact Statement

This work aims to support more reliable evaluation and deployment of coding agents by studying the maintainability of code produced by these systems. Our findings show that agent code can affect downstream maintenance, underscoring the need for benchmarks, review practices, and safeguards that evaluate coding agents beyond immediate task completion and support informed decisions about when and how to rely on agent code. These findings should not be interpreted as evidence against the use of coding agents broadly, but as a call for evaluations that better reflect their long-term effects on software development.

LLM/Agent Usage Disclosure

We used LLMs and coding agents at several stages of this work. During ideation, we used LLMs to search software engineering literature on maintainability. During experimentation, we used coding agents for debugging and to evaluate tools for computing maintainability metrics. All substantive research decisions, including the study design, benchmark construction criteria, metric selection, and interpretation of results were made by the authors. LLMs were not used to determine benchmark outcomes or compute the primary quantitative results, except where explicitly described for LLM-as-a-judge results.

References

Anthropic. Claude sonnet 4.5. URL https://www.anthropic.com/news/claude-sonnet-4-5.

Ardito, L., Coppola, R., Barbato, L., and Verga, D. A tool-based perspective on software code maintainability metrics: A systematic literature review. *Scientific Programming*, 2020(1):8840389, 2020. doi: https://doi.org/10.1155/2020/8840389. URL https://onlinelibrary.wiley.com/doi/abs/10.1155/2020/8840389.

Campbell, G. A. Cognitive complexity — an overview and evaluation. In *2018 IEEE/ACM International Conference on Technical Debt (TechDebt)*, pp. 57–58, 2018.

Chen, H., Li, C., and Li, J. Featbench: Evaluating coding agents on feature implementation for vibe coding. *arXiv preprint arXiv:2509.22237*, 2025.

Chen, J., Xu, X., Wei, H., Chen, C., and Zhao, B. Swe-ci: Evaluating agent capabilities in maintaining codebases via continuous integration, 2026. URL https://arxiv.org/abs/2603.03823.

Chowdhury, N., Aung, J., Shern, C. J., Jaffe, O., Sherburn, D., Starace, G., Mays, E., Dias, R., Aljubeh, M., Glaese, M., Jimenez, C. E., Yang, J., Ho, L., Patwardhan, T., Liu, K., and Madry, A. Swe-bench verified, 2024. URL https://openai.com/index/introducing-swe-bench-verified.

Cynthia, S. T., Muttakin, A., and Roy, B. Beyond bug fixes: An empirical investigation of post-merge code quality issues in agent-generated pull requests. *arXiv preprint arXiv:2601.20109*, 2026.

Deng, X., Da, J., Pan, E., He, Y. Y., Ide, C., Garg, K., Lauffer, N., Park, A., Pasari, N., Rane, C., Sampath, K., Krishnan, M., Kundurthy, S., Hendryx, S., Wang, Z., Bharadwaj, V., Holm, J., Aluri, R., Zhang, C. B. C., Jacobson, N., Liu, B., and Kenstler, B. Swe-bench pro: Can ai agents solve long-horizon software engineering tasks?, 2025. URL https://arxiv.org/abs/2509.16941.

Ebert, C., Cain, J., Antoniol, G., Counsell, S., and Laplante, P. Cyclomatic complexity. *IEEE Software*, 33(6):27–29, 2016. doi: 10.1109/MS.2016.147.

Gautam, D., Garg, S., Jang, J., Sundaresan, N., and Moghaddam, R. Z. Refactorbench: Evaluating stateful reasoning in language agents through code, 2025. URL https://arxiv.org/abs/2503.07832.

Halstead, M. H. *Elements of Software Science*. Elsevier North-Holland, New York, 1977.

Harding, W. and Kloster, M. AI Copilot code quality: 2025 look at long-term effects. Technical report, GitClear, 2025. URL https://www.gitclear.com/ai_assistant_code_quality_2025_research.

He, K. and Roy, K. Swe-adept: An llm-based agentic framework for deep codebase analysis and structured issue resolution, 2026. URL https://arxiv.org/abs/2603.01327.

Huang, H., Jaisri, P., Shimizu, S., Chen, L., Nakashima, S., and Rodríguez-Pérez, G. More code, less reuse: Investigating code quality and reviewer sentiment towards ai-generated pull requests, 2026. URL https://arxiv.org/abs/2601.21276.

Jimenez, C. E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O., and Narasimhan, K. R. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

Joshi, T., Chowdhury, S., and Uysal, F. Swe-bench-cl: Continual learning for coding agents, 2025. URL https://arxiv.org/abs/2507.00014.

Khandpur, K. Swe-bench multilingual, 2025. URL https://kabirk.com/multilingual.

Li, H., Zhang, H., and Hassan, A. E. The rise of ai teammates in software engineering (se) 3.0: How autonomous coding agents are reshaping software engineering, 2025. URL https://arxiv.org/abs/2507.15003.

Liu, S., Liu, F., Li, L., Tan, X., Zhu, Y., Lian, X., and Zhang, L. An empirical study on failures in automated issue solving, 2025. URL https://arxiv.org/abs/2509.13941.

Liu, Y., Widyasari, R., Zhao, Y., Irsan, I. C., Chen, J., and Lo, D. Debt behind the ai boom: A large-scale empirical study of ai-generated code in the wild, 2026. URL https://arxiv.org/abs/2603.28592.

MiniMax. Minimax m2.5. URL https://www.minimax.io/news/minimax-m25.

OpenAI. Gpt 5. URL https://openai.com/index/introducing-gpt-5.

OpenAI. Codex: AI Coding Agent. https://chatgpt.com/codex/, 2026. Accessed: 2026-04-25.

Orlanski, G., Roy, D., Yun, A., Shin, C., Gu, A., Ge, A., Adila, D., Sala, F., and Albarghouthi, A. Slopcodebench: Benchmarking how coding agents degrade over long-horizon iterative tasks. *arXiv preprint arXiv:2603.24755*, 2026.

Paul, D. G., Zhu, H., and Bayley, I. Investigating the smells of llm generated code, 2025. URL https://arxiv.org/abs/2510.03029.

Queirós, R. Codeskelgen-a program skeleton generator. 2013.

Riaz, M., Mendes, E., and Tempero, E. A systematic review of software maintainability prediction and metrics. In *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, ESEM '09, pp. 367–377, USA, 2009. IEEE Computer Society. ISBN 9781424448425. doi: 10.1109/ESEM.2009.5314233. URL https://doi.org/10.1109/ESEM.2009.5314233.

Ryoo, J., Fonseca, F., and Janzen, D. S. Teaching object-oriented software engineering through problem-based learning in the context of game design. In *2008 21st Conference on Software Engineering Education and Training*, pp. 137–144. IEEE, 2008.

Sun, X., Ståhl, D., Sandahl, K., and Kessler, C. Quality assurance of llm-generated code: Addressing non-functional quality characteristics. *Journal of Systems and Software*, pp. 112885, 2026.

Wang, S., Wang, Z., Ma, D., Yu, Y., Ling, R., Li, Z., Xiong, F., and Zhang, W. Codeflowbench: A multi-turn, iterative benchmark for complex code generation, 2026. URL https://arxiv.org/abs/2504.21751.

Wang, Z., Ling, R., Wang, C., Yu, Y., Wang, S., Li, Z., Xiong, F., and Zhang, W. Maintaincoder: Maintainable code generation under dynamic requirements, 2025. URL https://arxiv.org/abs/2503.24260.

Xu, J., Deng, K., Li, W., Yu, S., Tang, H., Huang, H., Lai, Z., Zhan, Z., Wu, Y., Zhang, C., Lei, K., Yao, Y., Lei, X., Zhu, W., Feng, Z., Li, H., Xiong, J., Li, D., Gao, Z., Wu, K., Xiang, W., Zhan, Z., Zhang, Y., Gong, W., Gao, Z., Wang, G., Xue, Y., Li, M., Xie, M., Zhang, X., Wang, J., Zhuang, W., Lin, Z., Wang, H., Zhang, Z., Zhang, Y., Zhang, H., Chen, B., and Liu, J. Swe-compass: Towards unified evaluation of agentic coding abilities for large language models, 2025. URL https://arxiv.org/abs/2511.05459.

Yang, J., Jimenez, C. E., Wettig, A., Lieret, K., Yao, S., Narasimhan, K. R., and Press, O. SWE-agent: Agent-computer interfaces enable automated software engineering. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. URL https://arxiv.org/abs/2405.15793.

Yu, K., Zhou, Z., Zeng, J., Wang, Y., Du, X., Yuan, Z., Liu, J., Zhou, Z., Wang, Y., Wang, C., and Peng, X. Does pass rate tell the whole story? evaluating design constraint compliance in llm-based issue resolution, 2026. URL https://arxiv.org/abs/2604.05955.

Zeng, A., Lv, X., Zheng, Q., Hou, Z., Chen, B., Xie, C., Wang, C., Yin, D., Zeng, H., Zhang, J., et al. Glm-4.5: Agentic, reasoning, and coding (arc) foundation models. *arXiv preprint arXiv:2508.06471*, 2025.

A. Synthetic Problem Statement Construction for PR_1

Figure A1 shows the prompt used to generate synthetic PR_1 problem statements. Figure A2 shows the prompt template used for solving PR_1.

B. Resolve Rate and Maintainability Metric Comparison on the Full Dataset

Table A1 shows the comparison of PR_2 resolution rates under HA and AA conditions.

Benchmark	Model	#Instances	$PR_2{}^{HA}$ Resolve	$PR_2{}^{AA}$ Resolve	Δ
SWE-Bench Verified	Claude	215	152 (70.7%)	149 (69.3%)	-1.4
	GPT	186	114 (61.3%)	108 (58.1%)	-3.2
	GLM	207	137 (66.2%)	131 (63.3%)	-2.9
	MiniMax	233	173 (74.25%)	162 (69.53%)	-4.72
SWE-Bench Multilingual	Claude	38	21 (55.3%)	21 (55.3%)	0.0
	GPT	36	17 (47.2%)	18 (50.0%)	2.8
	GLM	58	40 (69.0%)	32 (55.2%)	-13.8
	MiniMax	86	54 (65.9%)	52 (63.4%)	-2.5
SWE-Bench Pro	Claude	313	141 (45.1%)	116 (37.1%)	-8.0
	GPT	280	109 (38.9%)	111 (39.6%)	1.0
	GLM	397	179 (45.1%)	135 (34.0%)	-11.1
	MiniMax	395	171 (42.3%)	150 (38.0%)	-4.3
FeatBench	Claude	17	8 (47.1%)	9 (52.9%)	5.8
	GPT	17	9 (52.9%)	7 (41.2%)	-11.7
	GLM	42	18 (42.9%)	14 (33.3%)	-9.5
	MiniMax	40	19 (47.5%)	16 (40.0%)	-7.5

Table A1. **PR_2 resolution rates under HA and AA conditions.** Across most model-benchmark pairs, building on agent code drops the resolve rate relative to building on human code. To save cost, the Claude and GPT-5 runs on SWE-bench Pro, Multilingual, and FeatBench are restricted to instances where the PR_1 gate passed for both MiniMax and GLM-4.7.

C. Agent Execution and Model Details

C.1. Agent Execution

All experiments use SWE-Agent under a single shared configuration: a `step_limit` of 250 steps, a `cost_limit` of $3 USD per instance, and high reasoning effort across all models. For each agent-authored patch, we run the harness once to produce a patch. We adopt each benchmark's official evaluation harness, and for FeatBench we follow the harbor framework's open pull request (#1218).

C.2. Model link list

Table A2 presents the closed- and open-source models included in our benchmark.

Model	Provider	Reference
Closed-source models		
Claude 4.5 Sonnet	Anthropic	https://www.anthropic.com/claude
GPT-5	OpenAI	https://openai.com/index/introducing-gpt-5/
Open-source models		
GLM 4.7 FP8	Z.ai	https://huggingface.co/zai-org/GLM-4.7-FP8
MiniMax M2.5	MiniMax	https://huggingface.co/MiniMaxAI/MiniMax-M2.5

Table A2. Models evaluated in this work, grouped by access type. Closed-source models are accessed via provider APIs; open-source models are run via self-hosted vLLM.

D. LR Feature Details and LLM-as-a-Judge Details

D.1. Maintainability Metric Computation

We compute four maintainability metrics—cyclomatic complexity (CC), cognitive complexity (CogC), Halstead volume (HV), and logical lines of code (LLOC)—across all supported languages in our corpus. No single tool covers every metric across every language, so we combine multiple open-source libraries and one custom tool, selected to maximize per-language consistency. Table A3 lists each library's metric coverage. To fill a gap in Go tooling, we extend Maintidx, an existing Go library, into a custom `go-halstead` package that computes Halstead volume.

Library	Languages	CC	CogC	HV	LLOC
radon	Python	✓	—	✓	✓
complexipy	Python	—	✓	—	—
rust-code-analysis-cli	Rust, C, Java, JS, TS	✓	✓	✓	✓
go-halstead (custom)	Go	—	—	✓	✓
gocognit	Go	—	✓	—	—
lizard	Go, Ruby	✓	—	—	✓

Table A3. Metric coverage of the libraries used to compute maintainability metrics. ✓indicates supported, — indicates unsupported.

D.2. Patch Localization Features

The patch localization features measure where each chain's PR_2 patch edits. For each instance and each condition (HA, AA), we extract the set of files and the set of (file, function) pairs touched by the PR_2 patch. We parse each post-patch source file with an AST parser (Python's built-in `ast` module[1] for Python and `lizard`[2] for all other supported languages) and compare against the pre-patch state to identify which files and functions the agent added, edited, or removed. We then compute the Jaccard overlap between the HA and AA edit sets on the same instance. Given two sets A and B, the Jaccard overlap is:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}.$$

For each discordant instance we compute two Jaccards: J_i^{file} over the file sets and J_i^{func} over the (file, function) sets, comparing HA's PR_2 patch with AA's PR_2 patch on the same instance. A Jaccard of 1.0 means the two chains edited exactly the same set; a Jaccard of 0 means the two sets are disjoint.

D.3. PR_1 Behavioral Drift

Prior work on agent failure analysis typically looks at the trajectory and the final agent-produced patch to classify failures into various categories (Xu et al., 2025; Deng et al., 2025). Given our two-stage setup, we adapt this taxonomy to capture how Agent PR_1 diverges from Human PR_1. These divergences can silently propagate and cause failure during PR_2, as observed in Section 5. We develop a ten-category taxonomy that captures changes to the function's input contract, changes to its observable outputs and side effects, and edits that go beyond the stubbed target functions. Any of these can leave the downstream PR_2 task unresolved. Figure A3 shows the full system prompt. The prompt contains the definition of each category. We also reproduce the full definitions in Table A5. Figure A4 shows the user prompt and the output schema. We use DeepSeek V4 Pro as our LLM judge. We run it on 2,517 records, one for every instance in our four-model by four-benchmark grid.

D.4. PR_1 Error Compounding into PR_2

The PR_1 behavioral drift judge in Appendix D.3 identifies instances where Agent PR_1 diverges from Human PR_1. A PR_1 divergence does not automatically mean the chain fails. Two scenarios can neutralize it. First, the agent's PR_2 may rewrite the divergent function, replacing the drift with a different code block. Second, even when the drift survives, the benchmark's

[1] https://docs.python.org/3/library/ast.html
[2] https://github.com/terryyin/lizard

unit tests may not exercise it. The sharper question is therefore: did a PR_1 divergence survive into PR_2's final state, and did that survival cause the AA failure? We use a second LLM judge to answer this question. The judge runs only on the AA chain. The HA chain has no agent-introduced PR_1 divergence to compound. Figure A5 and Figure A6 show the full system and user prompt with the output schema. We use DeepSeek V4 Pro as the judge model. We exclude instances labeled `EQUIVALENT` or `STYLE` by the PR_1 behavioral drift judge because they have no divergence to compound, which leaves 2,153 instances.

Outcome	n	%
Drift survived	225	85.9
Partially corrected	19	7.3
Rewrote / corrected	18	6.9

Did the PR_1 drift survive PR_2? PR_2 leaves the agent's PR_1 drift unchanged on 85.9% of the 262 discordant instances — the drift almost always carries through.

Outcome	n	%
Yes (compounding-confirmed)	**54**	**20.6**
No (unrelated failure)	195	74.4
Unclear	12	4.6
Not applicable (passed)	1	0.4

Did the failure trace back to the drift? The surviving PR_1 drift directly causes the failing test on 20.6% of the 262 discordant instances. The remaining instances fail for unrelated reasons.

Table A4. **LLM-as-a-judge attribution on the HA-wins discordant cohort** (n=262 instances with a divergent agent PR_1; 30 PR_1-equivalent/style cases excluded). Drift survives PR_2 in 86% of cases; on 20.6% of the cohort the surviving drift directly traces to the failing test.

Category	Definition
Group A: functionally compatible	
EQUIVALENT	Same logic, control flow, and observable contract. Only cosmetic differences such as whitespace, comments, docstrings, or local-variable renames.
STYLE	Same observable contract and the same operations in the same order, expressed in a different surface syntactic form (e.g., list comprehension vs. loop, ternary vs. if/else, early return vs. nested).
ALGORITHM	Same observable contract, but the operations themselves differ in identity, count, or resource regime (e.g., iterative vs. recursive, added memoization, different short-circuit ordering).
Group B: behavioral divergence	
INPUT GATING	Agent added or removed an explicit input check that prevents the main body from executing on some inputs. The function exits early via a raise, an early return, or an assert.
SILENT FALLBACK	Agent silently substituted or coerced an input so that the main body still runs but on different data (e.g., `dict.get(k, default)`, `int(x)`, `x or default`).
ERROR HANDLING	Agent raises, catches, or wraps exceptions differently on a structurally equivalent code path (e.g., different exception type at the same raise site, propagating vs. catching).
RETURN SHAPE	Return value's type, structure, or layout differs (e.g., `None` vs. `[]`, scalar vs. tuple, different dict keys or layout).
EXTERNAL EFFECTS	Observable mutations or I/O differ (e.g., stdout or log calls added or removed, mutation of arguments, writes to instance or module state).
Group C: scope	
EXTRA EDITS	Target functions are behaviorally compatible, but the agent also modified non-target files or functions that were not mechanically required. Imports and helpers called from the target body do not count.
Group D: agent failure	
INCOMPLETE	Agent left at least one target stub in a non-functional or placeholder state (e.g., empty body, `raise NotImplementedError`, return paired with a `TODO` comment).

Table A5. The ten PR_1 behavioral drift categories. The judge picks exactly one primary category per instance.

Problem Statement Generation Prompt

```
You are creating a detailed docstring for a Lead Software Engineer hiring test. The candidate will receive ONLY the function signature and this docstring, and must implement the complete function.

***Full File Context***:
{full_file_content}

***Specific Function to Describe***:
{function_code}

***Function Name***: {function_name}

Create a comprehensive docstring that enables a senior engineer to implement this function correctly. Use standard docstring format with these sections:

**Format Requirements**:
- Brief summary (1-2 sentences) describing the function's purpose
- Args: List each parameter with type hints and detailed explanation
- Returns: Exact structure and type of return value
- Raises: All exceptions with triggering conditions
- Implementation Notes: Critical algorithm steps, validations, and logic flow (numbered list)
- Edge Cases: Non-obvious behaviors, special handling, database features, or conditional logic

**CRITICAL**: Include enough detail that the function can be implemented without seeing the original code. Focus on:
- Exact return value structure (not just "returns SQL" but "returns tuple of ([sql_string], params_list)")
- Key algorithmic steps in order
- Conditional logic and when it applies
- Error handling and validation checks

**Example of Good Docstring**:

Constructs SQL for combining multiple queries using set operators (UNION, INTERSECT, DIFFERENCE).

Args:
    combinator (str): Set operation type - 'union', 'intersect', or 'difference'
    all (bool): If True, adds 'ALL' modifier to UNION operations (ignored for other combinators)

Returns:
    tuple: (result, params) where:
        - result (list): Single-element list containing the combined SQL string
        - params (list): Flattened list of all parameters from combined queries

Raises:
    DatabaseError: If LIMIT/OFFSET used in subqueries when supports_slicing_ordering_in_compound is False,
                   or if ORDER BY used in subqueries when this feature is unsupported
    EmptyResultSet: If all combined queries are empty after filtering
    NotSupportedError: If the specified combinator is not supported by the database backend

Implementation Notes:
    1. Filter out empty queries from self.query.combined_queries
    2. If database doesn't support slicing/ordering in compounds, validate no LIMIT/OFFSET/ORDER BY in subqueries
    3. Synchronize values_select across queries: if one query has values_select and another doesn't,
       copy extra_select, values_select, and annotation_select to the empty one
    4. Compile each query with as_sql(), catching EmptyResultSet:
       - For UNION: skip empty queries and continue
       - For DIFFERENCE: skip only if parts already exist, otherwise re-raise
    5. Wrap each compiled SQL based on features:
       - If query has combinator and !supports_parentheses_in_compound: wrap in "SELECT * FROM (sql)"
       - If query has combinator and !supports_slicing_ordering_in_compound: wrap in "(sql)"
    6. Get combinator SQL from connection.ops.set_operators[combinator]
    7. Append ' ALL' if combinator is 'union' and all is True
    8. Join all SQL parts with combinator operator
    9. Flatten and extend params from all queries

Edge Cases:
    - Empty combined_queries list raises EmptyResultSet
    - Mixed empty/non-empty queries: empty ones are conditionally skipped based on combinator type
    - Nested combinators are wrapped differently based on supports_parentheses_in_compound
    - Column count must match across all combined queries (handled by values_select sync)

---

Now write a docstring of similar detail and structure for the target function. Return ONLY the docstring text, with no JSON formatting, code blocks, or extra commentary:"""
```

Figure A1. **Prompt used to generate synthetic PR_1 problem statements.** Given the full file context, target function code, and function name, the model is instructed to produce a standalone docstring that enables implementation without revealing the original function body. The prompt specifies the required sections: summary, arguments, returns, exceptions, implementation notes, and edge cases.

Example of synthetic problem statement

Task Overview:
Implement the following functions: `_cstack` in `astropy/modeling/separable.py`

Functions to Implement

File: `astropy/modeling/separable.py`

Function: `_cstack(left, right)`

Summary:
This function computes the result of the `&` operation between two models or coordinate matrices, effectively stacking their outputs horizontally.

Args:
- `left` (`astropy.modeling.Model` or `ndarray`): The left operand, which can be a model or a coordinate matrix representing outputs and inputs.
- `right` (`astropy.modeling.Model` or `ndarray`): The right operand, which can also be a model or a coordinate matrix.

Returns:
- `ndarray`: A 2D array resulting from the horizontal stacking of the coordinate matrices of the left and right operands.

Raises:
- `ModelDefinitionError`: Raised if the left and right operands do not have compatible dimensions for stacking.

Edge Cases:
- If both `left` and `right` are empty or have incompatible shapes, the function should raise a `ModelDefinitionError` indicating the mismatch in dimensions.
- If either `left` or `right` is a model with no outputs, the function should handle this gracefully by returning a matrix filled with zeros of appropriate shape.

Implementation Steps:
1. Compute the total number of outputs (`noutp`) by calling the `_compute_n_outputs` function with `left` and `right` as arguments.
2. Check if `left` is an instance of `Model`. If it is, call `_coord_matrix(left, 'left', noutp)` to get its coordinate matrix; otherwise, initialize a zero matrix of shape `(noutp, left.shape[1])` and fill the top part with the `left` matrix.
3. Similarly, check if `right` is an instance of `Model`. If it is, call `_coord_matrix(right, 'right', noutp)` to get its coordinate matrix; otherwise, initialize a zero matrix of shape `(noutp, right.shape[1])` and fill the bottom part with the `right` matrix.
4. Return the horizontally stacked result of `cleft` and `cright` using `np.hstack`.

Figure A2. **Prompt template used for solving PR_1.** The prompt is formatted in markdown and includes multiple components: a task overview, function summary, arguments, returns, raised exceptions, edge cases, and step-by-step implementation guidance.

You are an expert software engineer. You are given two implementations of the same set of target functions:
- HUMAN PR-1 (reference): the canonical base-state code that existed in the repository before stubbing. Treat this as the ground truth.
- AGENT PR-1: an LLM agent's attempt to fill in the same stubs.

The PR-0 stub preserves the original function signature and replaces only the body (e.g., with `pass` in Python). Both human and agent therefore work from a fixed signature, so signature-level changes are extremely rare by construction.

The user prompt is split into FOUR code blocks:
1. "HUMAN PR-1 - target functions": the human's implementation of the listed stub functions.
2. "AGENT PR-1 - target functions": the agent's implementation of the same stub functions.
3. "EXTRA HUMAN PR-1 EDITS - non-target functions or files": hunks the human's PR-1 also included that are NOT for the listed target functions. Often empty.
4. "EXTRA AGENT PR-1 EDITS - non-target functions or files": same shape for the agent. Use this to fire EXTRA_EDITS.

Compare block 1 vs block 2 to assess behavioral fidelity of the target implementation. Compare block 4 vs block 3 to assess whether the agent added gratuitous edits beyond what the human or the implementation required.
Your job is to classify how AGENT PR-1 differs from HUMAN PR-1 in behavioral fidelity. Pick EXACTLY ONE primary label below.

GROUP A - EQUIVALENCE LADDER (functionally compatible)
- EQUIVALENT: Same logic, same control-flow structure, same observable contract. Only cosmetic differences: whitespace, comments, docstrings, formatting, or LOCAL-VARIABLE renames. Class metadata constants that are never read (e.g., a __version__ bump that no caller observes) also fall here.

- STYLE: Same observable contract, same observable operations performed in the same order, expressed in a different surface syntactic form. The test: could a standard mechanical refactoring transform one into the other? Eligible refactorings include: * extract / inline method * list comprehension <-> loop + append * ternary expression <-> if/else * method chain <-> intermediate variables * match / case <-> if / elif chain * early-return-flat <-> structurally-nested * De Morgan's law on boolean conditions * branch reordering with corresponding branch-swap, ONLY when the swap is verifiable via De Morgan or a tautology - NOT when the swap relies on reasoning about the runtime domain of inputs. For example, `if not (x or y): A else: B` <-> `if x or y: B else: A` is verifiable by De Morgan and is STYLE. But `if x == 0: A else: B` <-> `if x > 0: B else: A` relies on `x` being non-negative across the input domain, which is NOT a tautology - this falls to ALGORITHM (different short-circuit ordering). * different error MESSAGE wording with the SAME exception TYPE * different log MESSAGE text when both versions still emit the same log call at the same point (the call exists in both; only the message string differs) The set of operations performed is identical; only the surface syntax differs.

- ALGORITHM: Same observable contract, but the operations performed differ in identity, count, or resource regime. Triggers: * recursive vs iterative (different call regime) * added memoization, caching, or other internal state held only within the function's local scope * different internal data structure for the same task (set vs list vs dict for a lookup) * one-pass vs multi-pass * loop-invariant code motion that changes how many times an operation runs * different short-circuit ordering that runs different work on different inputs, even when final outputs match on observed cases - INCLUDING branch-swaps that rely on input-domain reasoning rather than logical tautology * asymptotic complexity differences Complexity-class differences are one trigger but not the only one.

GROUP B - BEHAVIORAL DIVERGENCE (different observable contract)
These five labels classify WHICH DIMENSION of the observable contract differs.
- INPUT_GATING: Agent added or removed an explicit check on inputs or accessed data that prevents the main body from executing on some inputs. Triggers: * Added `if x is None: raise ValueError(...)` on top of human's direct usage. * Added `if not x: return None` or similar early-return guard. * Added `assert` statements on inputs. * Explicit length, type, membership, or null checks that raise or early-return. * Removed such a check that the human had. Defining property: on at least some inputs, the agent's version skips the main computation entirely (via raise, early return, or assert).

- SILENT_FALLBACK: Agent silently transformed inputs or accessed data in a way that prevents a downstream raise the other side would have hit, or substituted defaults / fallbacks for missing or malformed inputs. Triggers: * `dict.get(key, default)` substituted for `dict[key]` * `int(x)`, `str(x)`, or other type wrappers around inputs * `x or default` patterns * `try: ... except: pass` around an input read * Fallback values that mask errors the other side would have raised * Default substitution for missing attributes / keys Defining property: the main computation still runs, but with silently substituted or coerced data; no caller-observable signal announces the substitution.

- ERROR_HANDLING: Differences in how exceptions are raised, propagated, or wrapped along a structurally-equivalent code path. Triggers: * Different exception TYPE at the same raise site (e.g., both versions call `json.loads`, but agent catches and re-raises as `ValueError` where human lets `JSONDecodeError` propagate). * One side wraps a substantive operation (parse, network call, computation) in try/except while the other propagates. * One side uses `raise X from e` and the other does not, or uses a different wrapper type. ERROR_HANDLING does NOT fire when the agent added an input check (-> INPUT_GATING) or when the agent silently substituted a fallback for an input (-> SILENT_FALLBACK). It fires for differences in what happens when exceptions are raised or propagated, not for differences in input pre-checking.

- RETURN_SHAPE: The return value's type, structure, or layout differs. Triggers: * None vs []; scalar vs tuple; generator vs list * Different dict keys or layout * Different truthy/falsy convention * (Rare residual) The agent deviated from the provided signature: parameter renames, parameter additions or removals, changed default values, or changed type annotations. Since PR-0 fixes the signature, this is expected to be near-zero, but route here when it occurs. * Class metadata constants that ARE read reflectively by callers (e.g., __version__ exposed in an API response) -> RETURN_SHAPE if the value differs. If never read -> EQUIVALENT.

- EXTERNAL_EFFECTS: Externally-observable mutations or I/O differ, or their ordering differs. Triggers: * I/O: stdout, stderr, files, network, database * Mutation of arguments (e.g., `arr.sort()` vs `sorted(arr)`) * Mutation of instance state (`self.x = ...`), class state, or module state * Cache writes that other code can read * Log call ADDED or REMOVED (the call itself is the trigger; a log call that exists in both versions with only different message text is STYLE, not EXTERNAL_EFFECTS) * Different ordering of any of the above where the order is observable Local variable assignments and internal data transformations that don't escape the function do NOT count. Boundary with SILENT_FALLBACK: if the agent reads an input and silently substitutes a default, that's SILENT_FALLBACK. If the agent writes back to the input or to shared state, the mutation escapes -> EXTERNAL_EFFECTS. Boundary with ALGORITHM: agent adds memoization. If the cache is a local variable discarded on return -> ALGORITHM. If the cache is module-level or attached to `self` so future calls see it -> EXTERNAL_EFFECTS. Boundary with STYLE: different error message text with the same exception type is STYLE, UNLESS the message leaks internal state the other side didn't expose -> EXTERNAL_EFFECTS (information escapes through the exception channel). Same rule for log message text: STYLE for wording, EXTERNAL_EFFECTS only if the message leaks state.

GROUP C - SCOPE (right code, extra place)

- EXTRA_EDITS: Agent edited the listed target functions correctly enough on the behavioral axes (no Group B label fires), AND ALSO modified one or more non-target functions or files that were not required. Concretely: the "EXTRA AGENT PR-1 EDITS" block (block 4) contains hunks that have no counterpart in the "EXTRA HUMAN PR-1 EDITS" block (block 3) and that are not mechanically required by the target-function implementation. Mechanically required means: * imports the target function body uses * helper functions the target body calls * type definitions the target body references An agent-introduced helper that is called from a target function is NOT EXTRA_EDITS - it is mechanically required collateral.

GROUP D - AGENT FAILURE

- INCOMPLETE: Agent left at least one target stub in a non-functional or trivially-placeholder state. Patterns include: * Empty body: `pass` in Python; `{}` in Java/Go/Rust/C/JS/TS * Explicit not-implemented markers: `raise NotImplementedError`, `throw new UnsupportedOperationException`, `panic!("unimplemented")`, `todo!()`, `unimplemented!()`, `fmt.Errorf("not implemented")` * Any exception / panic / error whose message contains "not

TIEBREAKER
If multiple labels apply, pick the FIRST one that matches in this priority order:
INCOMPLETE > INPUT_GATING > SILENT_FALLBACK > ERROR_HANDLING > RETURN_SHAPE > EXTERNAL_EFFECTS > EXTRA_EDITS > ALGORITHM > STYLE > EQUIVALENT GENERAL RULES

GENERAL RULES

- Judge from the code alone. Do not speculate about agent quality or intentions beyond what the code shows.
- Pick exactly one primary label. - If multiple labels apply, walk the tiebreaker top to bottom and pick the first one that fits.
- "Observable" means: a caller could, in principle, write a test that distinguishes the two implementations. Apply this standard uniformly across labels.
- secondary_axes: include a label only if there is INDEPENDENT code evidence that the label fires on its own merits - not because the label "could conceivably also apply" or "is a close runner-up to the primary." Typical secondary_axes is empty or has 1 entry; cases with 2+ are rare. A label in secondary_axes should be traceable to a distinct hunk or change that the primary label doesn't already cover.

DECISION RULES FOR SPECIFIC LABEL PAIRS

- COSMETIC vs NOT COSMETIC: Identifier renames of LOCAL variables, comments, docstrings, whitespace alone -> EQUIVALENT.

- STYLE vs ALGORITHM: Apply the mechanical-refactoring test. If some sequence of the listed standard refactorings can transform one into the other without changing the set of operations performed AND without relying on runtime-domain reasoning, it's STYLE. If the agent has changed the operations themselves (different call regime, different data structure, different number of passes, added internal state, changed short-circuit behavior, or branch-swaps that depend on input-domain properties), it's ALGORITHM.

- STYLE vs ERROR_HANDLING: Different error MESSAGE wording with the SAME exception TYPE -> STYLE. Different exception TYPE, or one side raises while the other does not -> ERROR_HANDLING. If a divergent error message leaks internal state the other side didn't expose, treat as EXTERNAL_EFFECTS.

- INPUT_GATING vs SILENT_FALLBACK - the body-execution test: After the agent's added handling, does the function still execute the main computation?
  * YES, with substituted / coerced data -> SILENT_FALLBACK
  * NO, the function exits early (return, raise, assert) before the main body runs -> INPUT_GATING
For a single code change, these are mutually exclusive. They can co-occur in secondary_axes only when the agent introduces TWO separate changes - one that short-circuits and one that substitutes - at different points in the function.

- INPUT_GATING / SILENT_FALLBACK vs ERROR_HANDLING:
  * If the agent added or removed handling on the input that either short-circuits the body or silently substitutes data flowing into it, it's INPUT_GATING or SILENT_FALLBACK.
  * ERROR_HANDLING fires only when the structural code path is the same but the exception TYPE raised differs at the same point, or when the agent wraps internal logic in try/except where the human propagates (pure error suppression / transformation, not input pre-checking).

- INPUT_GATING / SILENT_FALLBACK vs EXTERNAL_EFFECTS: Input handling gates or transforms data flowing into the body. External effects are what the body does that leaks past return. Reading an input and substituting a default -> SILENT_FALLBACK. Writing back to the input or to shared state -> EXTERNAL_EFFECTS.

- RETURN_SHAPE: Literal changes that affect what is returned belong here. Literal changes that affect iteration scope or branching (`range(10)` -> `range(20)`, `if x < 5` -> `if x < 10`) belong to the appropriate behavioral label (INPUT_GATING / SILENT_FALLBACK / EXTERNAL_EFFECTS / ALGORITHM) rather than RETURN_SHAPE.

- EXTRA_EDITS excludes mechanically-required collateral. If the human's PR-1 (block 3) also contains an equivalent extra hunk, the agent's matching hunk is shared collateral, not EXTRA_EDITS. - EXTRA_EDITS vs behavioral labels: Extra edits outside the listed target set, while the target functions themselves are behaviorally equivalent -> EXTRA_EDITS. If the target functions ALSO have behavioral divergence (Group B), the behavioral label wins per the tiebreaker, and EXTRA_EDITS goes to `secondary_axes` if there is independent evidence of extra edits.

Respond ONLY with valid JSON in the exact schema shown in the user message. No prose, no markdown fences."""

Figure A3. **System prompt for the PR_1 behavioral drift judge.** Defines the ten-category taxonomy.

```
## Target functions (stub set):
    f"{targets_summary}

## HUMAN PR-1 (reference) — target functions
    {human_target_block}

## AGENT PR-1 — target functions
    {agent_target_block}

## EXTRA HUMAN PR-1 EDITS — non-target functions or files
    {human_extra_block}

## EXTRA AGENT PR-1 EDITS — non-target functions or files
    {agent_extra_block}

## Output
Respond ONLY with this JSON (no prose, no markdown fence):
    {
        "pr1_diff_summary": "<2-3 sentences stating the factual differences between Human PR-1 and Agent PR-1>
        "pr1_fidelity": "<one of: EQUIVALENT, STYLE, ALGORITHM, INPUT_GATING, SILENT_FALLBACK, ERROR_HANDLING,
RETURN_SHAPE, EXTERNAL_EFFECTS, EXTRA_EDITS, INCOMPLETE>
        "pr1_fidelity_reason": "<2-3 sentences citing specific code differences that justify the primary label. If multiple labels could apply, briefly note
which competing labels you considered and why the chosen one wins under the tiebreaker.>
        secondary_axes": ["<list of other labels from the same set that fire on independent code evidence; empty list if none>"]
    }
```

Figure A4. **User prompt for the PR_1 behavioral drift judge.** Defines the output schema.

```
You are an expert software-engineering analyst. You analyze a two-step change an autonomous agent made to a repository to resolve a GitHub issue:
    PR-1 -- the agent's FIRST step: it implemented/filled in the target function bodies (which started as empty stubs).
    PR-2 -- the agent's SECOND step, applied ON TOP of PR-1: a follow-up fix.

A separate prior review compared the agent's PR-1 against the human's reference PR-1 and identified ONE specific way the agent's PR-1 DIVERGED
from the human implementation (the "PR-1 divergence"). You are given that divergence.

Your job has two parts:
    Q1 SURVIVAL  -- did that PR-1 divergence survive into the final code after PR-2, or did PR-2 correct it?
    Q2 CAUSATION -- if the run failed, did the surviving divergence cause (or contribute to) the failure?
You are given the test outcome. Use it for Q2 ONLY. Q1 (survival) is a pure code-relationship question -- determine it from the two patches alone, NOT
from whether tests passed.
DEFINITIONS
divergence_location_in_pr2 -- where PR-2's edits fall relative to the divergent code PR-1 wrote:
    untouched  -- PR-2 does not edit the function/code carrying the divergence.
    edited     -- PR-2 changes that function but does not fully replace it.
    rewritten  -- PR-2 replaces/rewrites that function wholesale.

divergence_status -- is the specific divergence present in the FINAL code?
    survived            -- the divergent behavior is still present after PR-2.
    partially_corrected -- PR-2 addressed part of it; some of it remains.
    corrected           -- PR-2 removed/fixed the divergence; it is gone. (If divergence_location_in_pr2 == untouched, the divergence necessarily
    SURVIVED -- PR-2 changed nothing there.)

pr2_modifies_pr1_behavior -- does PR-2 CHANGE validation, default-value, or error-handling logic that PR-1 introduced? none | validation | default |
error_handling | multiple
pr2_relies_on_pr1_contract -- does PR-2 ASSUME (use, without changing) a return shape or state variable that PR-1 introduced? none | return_shape |
state_variable | both

failure_linked_to_divergence -- did the surviving divergence cause the failure?
    yes            -- a failing test fails BECAUSE of the divergence: the failing assertion / traceback traces to the divergent behavior.
    no             -- the run failed, but for a reason unrelated to the divergence.
    unclear        -- the run failed; cannot tell if the divergence is responsible.
    not_applicable -- the run PASSED; there is no failure to attribute.

Ground every decision in specific evidence. For Q2 you MUST cite a concrete failing-test name or traceback line AND the specific divergent code; if you
cannot, use `unclear`.

Respond with a SINGLE JSON object and nothing else:
{
    "pr1_divergence_restated": "<1 sentence: the PR-1 divergence, in your words>",
    "pr2_summary": "<1-2 sentences: what PR-2 changed>",
    "divergence_location_in_pr2": "untouched|edited|rewritten",
    "divergence_status": "survived|partially_corrected|corrected",
    "pr2_modifies_pr1_behavior": "none|validation|default|error_handling|multiple",
    "pr2_relies_on_pr1_contract": "none|return_shape|state_variable|both",
    "failure_linked_to_divergence": "yes|no|unclear|not_applicable",
    "failure_mechanism": "<1-3 sentences if linked=yes, else empty>",
    "link_evidence": "<failing-test/traceback line + divergent code line, else empty>",
     "confidence": "high|medium|low"
}
```

Figure A5. **System prompt for the PR_1/PR_2 compounding judge.**

```
## Issue / problem statement
{problem_statement}

## PR-1 divergence (from the prior PR-1-vs-human review)
divergence type: {pr1_fidelity_label}
what the agent's PR-1 did: {pr1_diff_summary}
detail: {pr1_fidelity_reason}

## PR-1 -- the agent's first change (fills the stubs)
```diff
    {pr1_patch}

PR-2 -- the agent's second change, applied on top of PR-1
    {pr2_patch}

Test outcome
    {test_outcome_block}

Assess Q1 (survival) and Q2 (causation) as instructed.

**The `{test_outcome_block}` has two variants** — pick whichever you want to show in the figure (or show both side-by-side):

**Resolved (passed):**
This chain PASSED all tests (all FAIL_TO_PASS and PASS_TO_PASS tests pass). There is no failure, set failure_linked_to_divergence = not applicable
and leave failure_mechanism / link_evidence empty.

**Failed:**
This chain FAILED.
    - FAIL_TO_PASS: {n_f2p_pass} passed, {n_f2p_fail} failed
    - PASS_TO_PASS: {n_p2p_pass} passed, {n_p2p_fail} failed
    - failing FAIL_TO_PASS tests: {f2p_fail_names}
    - failing PASS_TO_PASS tests: {p2p_fail_names}

Test output (tail)
    {test_output_tail}      # last 10,000 chars of test_output.txt
```

Figure A6. **User prompt for the PR_1/PR_2 compounding judge** Defines the output schema.
```