

Is Agent Code Less Maintainable Than Human Code?

Shaswat Patel^{1*}, Betty Li Hou^{1*}, Arun Purohit¹, Kai Xu¹, Jane Pan¹, He He¹, Valerie Chen²

¹New York University ²Carnegie Mellon University

{spp9399, betty.li.hou}@nyu.edu

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 coding agents not only by immediate task resolution but also by 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, 2026a). Model capabilities on coding tasks have also improved rapidly, with state-of-the-art agents solving a growing share of repository-level benchmarks (OpenAI, 2026b). However, a patch that passes tests and is functionally correct may introduce structural complexity, brittle abstractions, or

confusing implementation choices that only surface as failures when the code is extended or modified in downstream tasks (Figure 1).

This dimension of code quality is captured by *maintainability*, or how well the code can be modified and extended over time. 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 of a project (Riaz et al., 2009; Ardito et al., 2020). The adoption of coding agents raises questions about whether code authored by agents is as maintainable as human-authored code. Recent work has measured agent code maintainability using these static metrics (Wang et al., 2025; Orlanski et al., 2026), but does not compare it with human code. Moreover, static metrics can often be confounded by codebase conventions and context (Zhang et al., 2013).

To this end, we present **CodeThread**, a framework for controlled experiments to compare the maintainability of agent- and human-written code based on how well an agent can build on each. CodeThread transforms single-step, repository-level coding tasks (e.g., instances from SWE-Bench (Jimenez et al., 2024)) into two-step tasks comprising an initial implementation task and a dependent downstream task. CodeThread then constructs 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 use CodeThread to evaluate four frontier models—Claude 4.5 Sonnet (Anthropic), GPT-5 (OpenAI), GLM 4.7 (Zeng et al., 2025; Z.ai), and MiniMax 2.5 (MiniMax)—on four software

*Equal contribution.

Code and data available at: <https://github.com/shaswatpatel123/CodeThread>

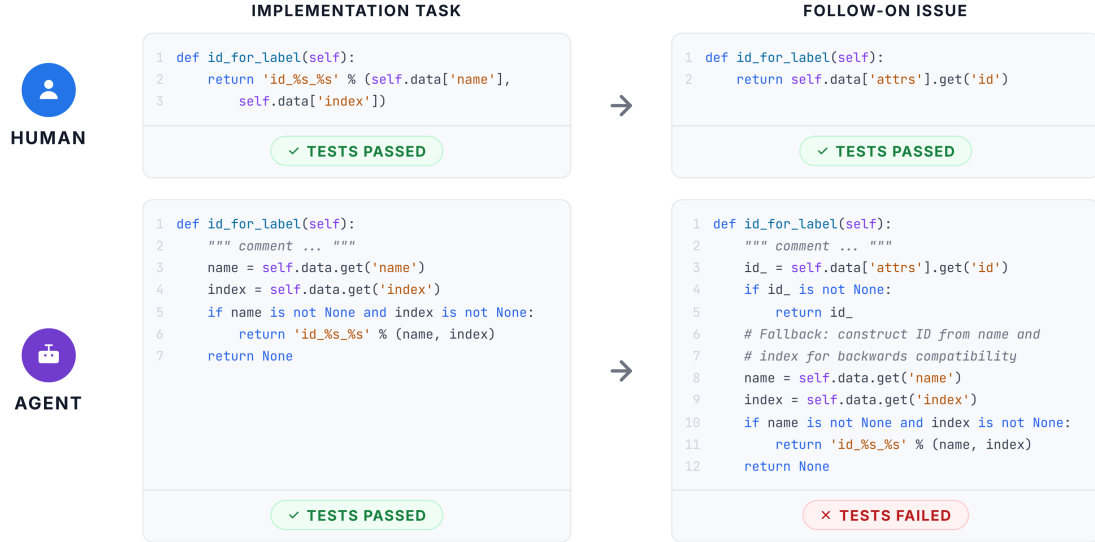


Figure 1: **Two pieces of code both being functionally correct does not mean they are equally maintainable.** An example instance where both the human and agent code pass the initial implementation task’s tests, yet when an agent makes a subsequent change to both, the version built on human code passes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

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 predictors of cases where the downstream task fails on agent code but succeeds on human code are 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 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 per-

formance.

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

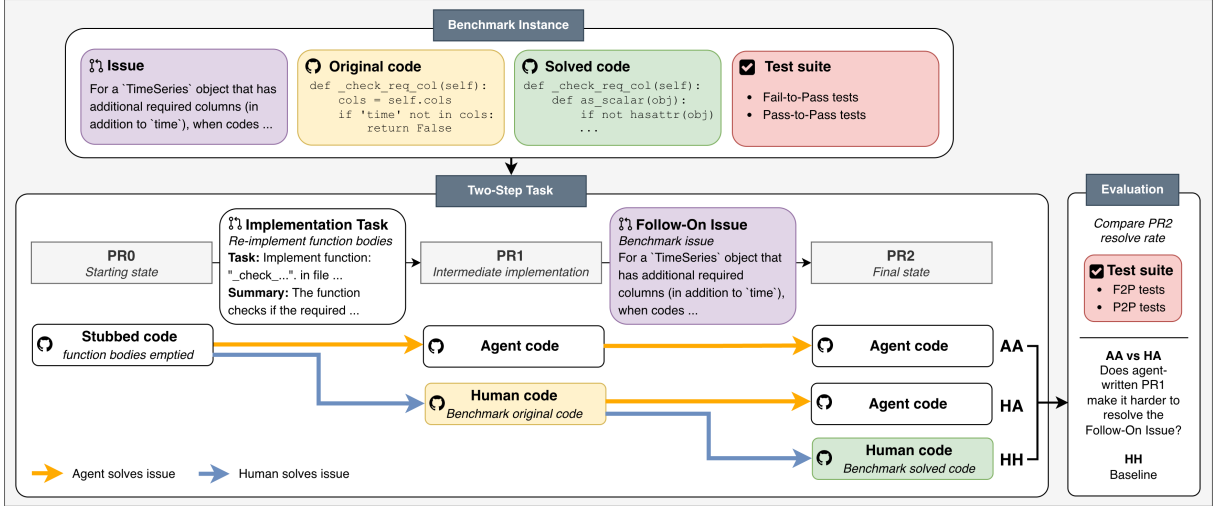


Figure 2: **CodeThread framework.** From the original benchmark instance, we construct a two-step task—an Implementation Task followed by a Follow-On Issue—producing three code states (PR₀, PR₁, and PR₂) and three conditions: AA (agent performs both steps), HA (human performs the Implementation Task, 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.

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 and 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 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 maintainability directly through downstream issue-resolution perfor-

mance. 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, as performed in 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 Feat-Bench (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 PR_0 , PR_1 , and 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. This issue is provided to the agent, which implements the requested functionality to produce the agent PR_1 . 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 the agent PR_1 reflects the benchmark’s initial code state rather than a partial or complete solution to the Follow-On Issue; accordingly, the target methods are functionally equivalent across the two PR_1 versions. Although agents 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 PR_1 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 PR_1 implementation. We evaluate whether the 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.

We focus on the maintainability costs that agents incur when building on prior code, comparing the outputs from the AA versus HA scenarios, as a growing fraction of developers use AI agents to extend existing codebases (Li et al., 2025). The AH setting of a human performing the follow-on task on top of agent code would offer a complementary human-centric view (by comparing AH versus AA), but would require 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 data collection from humans.

3.3 Step 3: Evaluate agent vs human code

We assess the final HA and AA outputs, PR_2 , using resolve rate: the percentage of instances in which the code successfully resolves the original benchmark issue after both editing steps. 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),

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 (PR₁, PR₂), and a breakdown of task types: bug fixes (BF), feature implementation (FI), and refactoring (RF).

and FeatBench (Chen et al., 2025). The four benchmarks were selected to 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. We use the native labels of each SWE-Bench Pro instance, and treat 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 further details in Appendix C.

4.2 Results

Overall findings. Table 2 shows the PR₂ resolve rates under HA and AA conditions. The number of instances that pass the PR₁ 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 a 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 trace the source of the gap between HA and AA in Section 5.

We also observe four cases where AA matches or exceeds HA performance. These should be considered in light of benchmark-specific baseline performance and sample size. GPT-5 shows small positive gaps on SWE-Bench Multilingual (+3.1) and SWE-Bench Pro (+0.8), but these coincide with lower HA baseline performance on the benchmarks, suggesting that the gap may reflect a weaker ceiling rather than greater robustness to agent code. The clearest positive result, Claude 4.5 Sonnet on FeatBench (+6.2), occurs on our smallest overlap set of 16 instances, where a small number of cases can substantially shift the observed gap.

Per-model patterns. Table 2 shows the HA-vs-AA gap across models. GLM shows the largest and most consistent effect of PR₁ authorship, with AA underperforming HA 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 mixed results. GPT-5 has the smallest gaps on SWE-Bench variants, however, we see a 12.5% drop on FeatBench, 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 show that refactoring and multi-file edits are harder for coding agents than localized fixes (Xu et al., 2025; Gautam et al., 2025).

Benchmark	Resolve Rate % ($\uparrow$)											
	Claude			GPT			GLM			MiniMax		
	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 PR₂ 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). PR₂ 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.

5 Why does AA underperform HA?

We observe that agents building on agent code generally perform worse than on human code. To investigate the drivers of this gap, we focus on discordant instances—i.e., those where HA and AA yield different outcomes on the task. Across all four models and benchmarks, there are 454 discordant instances. 64.3% of these are HA wins (HA passes while AA fails), while 35.7% are AA wins (AA passes while HA fails). After excluding instances with incomplete features, our analysis includes 405 instances (64.7% HA wins, 35.3% AA wins). We extract features from PR₁ and PR₂, detailed in Section 5.1, and fit a logistic regression model to identify which features most strongly predict whether HA or AA won in a discordant instance.

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 the files and functions edited, behavioral drift labels to capture differences 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.

Static maintainability metrics on PR₁ and PR₂.

Our first feature set comprises four static proxies for maintenance cost based on software engineering literature (Riaz et al., 2009; Ardito et al., 2020), spanning two dimensions of code quality. The first dimension is *structural complexity*, for which 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*, for which 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 AA–HA delta at PR₁ and PR₂.

Patch localization features on PR₂.

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 and Roy, 2026; Liu et al., 2025). We focus on PR₂ because the PR₁ prompt explicitly lists the files and functions each chain must edit. The PR₂ problem statement does not, and the agent can freely choose which files and functions to edit. We measure this with two Jaccard overlaps: one

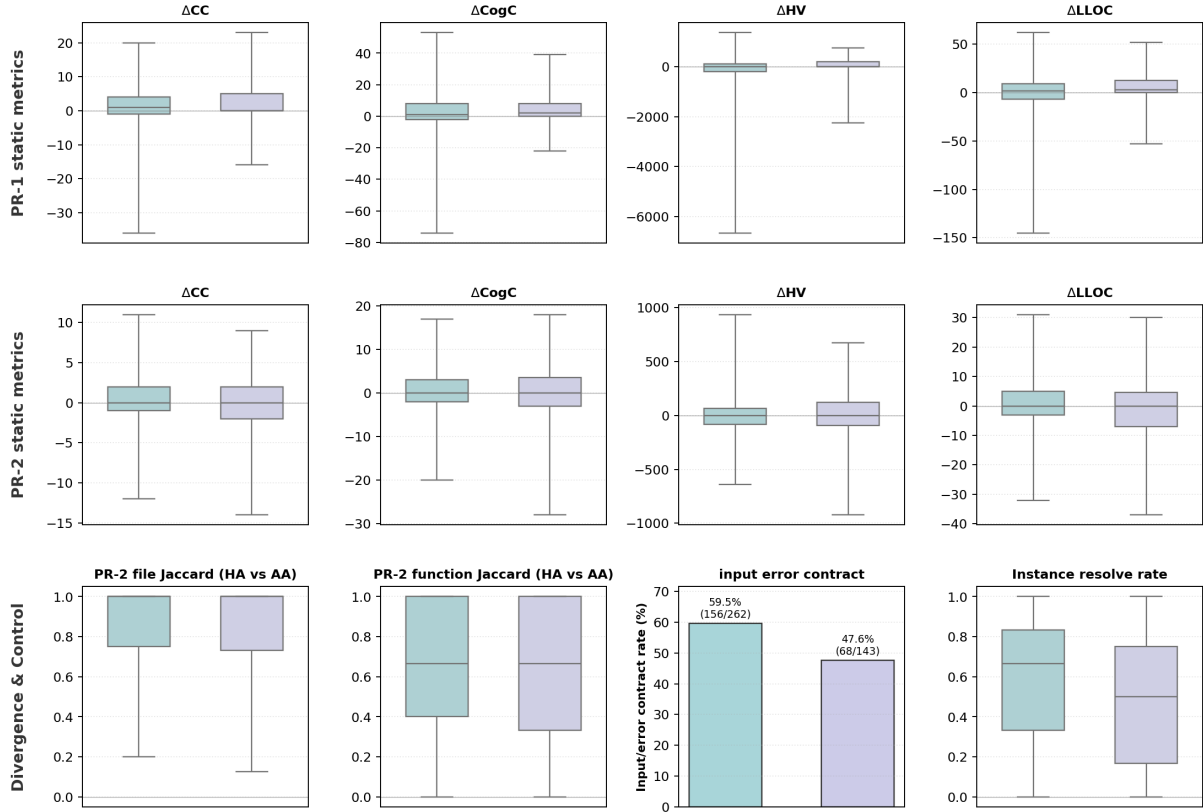


Figure 3: **HA vs AA-only wins are differentiated by behavioral drift, instance difficulty, and downstream code-size change.** Panels compare instances resolved only in HA (teal) versus in AA (lavender); whiskers show the 5th/95th percentiles. Most static maintainability metrics have similar distributions,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

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.

Behavioral drift labels on PR_1 . The third feature set captures differences between agent and human PR_1 s not reflected in aggregate static metrics. We develop a ten-category taxonomy for forms of behavioral drift, detailed in Appendix D.3, and use Deepseek-V4 Pro (DeepSeek-AI et al., 2026) as an LLM-judge to label instances by category. We are particularly interested in cases where the agent’s PR_1 alters input-validation or error-handling behavior, which we define as *input/error contract* (IEC). From our observations, differences in exception types or swapped defaults can silently creep into the codebase. Agents may add an extra input-gating check that raises `ValueError` or use `getattr` with `None` as the default, while the human PR_1 would raise an error. In both cases, neither static metrics nor patch location features would distinguish the agent from the human code. Appendix D.7 shows

five such cases contrasting the human and agent implementations of PR_1 and PR_2 . As such, we look at whether or not the behavioral drift was labeled as one of three IEC categories—Input Gating, Silent Fallback, and Error Handling—setting the feature to 1 if so, and 0 otherwise. To verify LLM-judge accuracy, we manually check a subset of 50 LLM-judge labels (Appendix D.5).

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 winning. We define $Y_i = 1$ when HA resolves and AA fails on instance

Feature	OR (95% CI)	<i>p</i>
$\Delta\text{CC}_{\text{PR}_1}$	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 PR₁ and PR₂ features on HA-to-AA Odds Ratio (OR).** 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 PR₁, AA over-editing at PR₂ relative to HA, and easier instances each raise the odds that only HA resolves.

i and $Y_i = 0$ otherwise. The model is defined as:

$$\begin{aligned} \text{logit } \Pr(Y_i = 1 \mid X_i) = & \beta_0 + \beta_{\text{IEC}} \text{IEC}_i + \beta_J^\top \mathbf{J}_i \\ & + \beta_1^\top \Delta_i^{\text{PR}_1} + \beta_2^\top \Delta_i^{\text{PR}_2} \\ & + \beta_D^\top \mathbf{D}_i + \beta_{\text{LOO}} \text{LOO}_i \end{aligned}$$

Static maintainability metrics are denoted with $\Delta_i^{\text{PR}_1}$ and $\Delta_i^{\text{PR}_2}$ for each of four AA–HA metric deltas (CC, CogC, HV, LLOC) at PR₁ and PR₂. For fault localization, the HA-vs-AA file and function Jaccards on the PR₂ patches are denoted by $\mathbf{J}_i = (J_i^{\text{file}}, J_i^{\text{func}})$. PR₁ behavioral drift is denoted by $\text{IEC}_i \in \{0, 1\}$, an input/error-contract indicator on the agent’s PR₁. 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 $\text{OR} > 1$ therefore tilt the outcome toward HA, and $\text{OR} < 1$ 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 wins and AA wins.

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 winning: IEC, $\Delta\text{LLOC}_{\text{PR}_2}$, and Instance resolve rate. When agent PR₁ alters input-validation or error-handling behavior, the odds of HA winning are $1.83\times$ that of AA, and when AA’s PR₂ 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 $\chi^2(18) = 36.1$, $p = 0.007$) suggesting that these features are informative signals of downstream success, but do not fully explain or predict it.

Figure 3 corroborates these results as static maintainability metrics show near-identical distributions on HA wins and AA wins, while IEC drift, $\Delta\text{LLOC}_{\text{PR}_2}$, and instance resolve rate show the clearest visual separation between the two groups. To verify the LLM-judge labels and assess robustness to the choice of judge, we re-ran the PR₁ behavioral drift judge with Kimi K2.6 (Kimi) and re-fit the same model. This yielded the same three significant features with comparable odds ratios (IEC: OR = 1.70, $p = 0.039$; see Appendix D.6 for further details).

Does PR₁ drift cause the unresolved outcome in PR₂? The IEC coefficient in Table 4 establishes the population-level signal, but population-level evidence does not tell us whether the drift was actually the cause of failing to resolve a given instance. We further investigate the cause of failures in the AA scenario, focusing on IEC for per-instance attribution as the only significant predictor in our model that identifies a specific, code-level mechanism. $\Delta\text{LLOC}_{\text{PR}_2}$ is a summary statistic and instance difficulty is a control variable, neither of which corresponds to a verifiable property of the agent’s PR₁.

We consider two scenarios that could decouple IEC drift from the downstream failure. First, the agent may rewrite the function containing the PR₁ input/error-contract divergence in PR₂, removing the drift before evaluation. Second, the drift may persist into PR₂, but the failing test may be evaluating a different part of the code. We use an LLM-judge (DeepSeek-V4 Pro) to attribute whether the failing test falls under either of these scenarios. The LLM-judge is given the PR₁ behavioral drift label with the generated rationale, the AA PR₂ patch, and the AA test outcome. It judges whether the divergence persisted into PR₂, and whether the fail-

ing test can be traced back to the divergence.

Table A4 reports the breakdown. The first scenario is rare: PR_2 rewrites the divergent function on only 6.9% of instances, while the drift is unchanged in 85.9% of instances. On 20.6% of instances, the drift is judged to both persist and directly relate to the failing test case. 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 provided in Appendix D.4.

6 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.

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. However, it may introduce selection bias, since restricting the analyzed instances to where the agent did not solve the Follow-On Issue at PR_1 may not represent the agent’s natural behavior across the full benchmark. Nonetheless, the filter retains a large majority of instances, and we verify that cases where the model satisfies Fail-to-Pass tests in the Implementation Task are often incidental.

Third, as discussed in Section 3.2, 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, and provide a more human-centric analysis of code maintainability. However, constructing this condition at scale would require recruiting developers to author PR_2 across all 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.

Ethical Considerations

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 the long-term effects on software development.

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 by NSF IIS-2340345, NSF CAREER Award 2145542, and Coefficient Giving (formerly Open Philanthropy), and in part by the NYU IT High Performance Computing resources, services, and staff expertise.

References

- Anthropic. [Claude sonnet 4.5](#).
- Luca Ardito, Riccardo Coppola, Luca Barbato, and Diego Verga. 2020. [A tool-based perspective on software code maintainability metrics: A systematic literature review](#). *Scientific Programming*, 2020(1):8840389.
- G. Ann Campbell. 2018. Cognitive complexity: An overview and evaluation. In *2018 IEEE/ACM International Conference on Technical Debt (TechDebt)*, pages 57–58.
- Haorui Chen, Chengze Li, and Jia Li. 2025. Featbench: Evaluating coding agents on feature implementation for vibe coding. *arXiv preprint arXiv:2509.22237*.
- Jialong Chen, Xander Xu, Hu Wei, Chuan Chen, and Bing Zhao. 2026. [Swe-ci: Evaluating agent capabilities in maintaining codebases via continuous integration](#). *Preprint*, arXiv:2603.03823.
- Neil Chowdhury, James Aung, Chan Jun Shern, Oliver Jaffe, Dane Sherburn, Giulio Starace, Evan Mays, Rachel Dias, Marwan Aljubeh, Mia Glaese, Carlos E. Jimenez, John Yang, Leyton Ho, Tejal Patwardhan, Kevin Liu, and Aleksander Madry. 2024. [Swe-bench verified](#).
- Shamse Tasnim Cynthia, Al Muttakin, and Banani Roy. 2026. Beyond bug fixes: An empirical investigation of post-merge code quality issues in agent-generated pull requests. *arXiv preprint arXiv:2601.20109*.
- DeepSeek-AI, Anyi Xu, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, Chenchen Ling, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chengyu Hou, Chenhao Xu, Chenze Shao, Chong Ruan, Conner Sun, and 300 others. 2026. [Deepseek-v4: Towards highly efficient million-token context intelligence](#). *Preprint*, arXiv:2606.19348.
- Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, Karmini Sampath, Maya Krishnan, Srivatsa Kundurthy, Sean Hendryx, Zifan Wang, Vijay Bharadwaj, Jeff Holm, Raja Aluri, Chen Bo Calvin Zhang, and 3 others. 2025. [Swe-bench pro: Can ai agents solve long-horizon software engineering tasks?](#) *Preprint*, arXiv:2509.16941.
- Christof Ebert, James Cain, Giuliano Antoniol, Steve Counsell, and Phillip Laplante. 2016. [Cyclomatic complexity](#). *IEEE Software*, 33(6):27–29.
- Dhruv Gautam, Spandan Garg, Jinu Jang, Neel Sundaresan, and Roshanak Zilouchian Moghaddam. 2025. [Refactorbench: Evaluating stateful reasoning in language agents through code](#). *Preprint*, arXiv:2503.07832.
- Maurice H. Halstead. 1977. [Elements of software science](#).
- William Harding and Matthew Kloster. 2025. [AI Copilot code quality: 2025 look at long-term effects](#). Technical report, GitClear.
- Kang He and Kaushik Roy. 2026. [Swe-adept: An llm-based agentic framework for deep codebase analysis and structured issue resolution](#). *Preprint*, arXiv:2603.01327.
- Haoming Huang, Pongchai Jaisri, Shota Shimizu, Lingfeng Chen, Sota Nakashima, and Gema Rodríguez-Pérez. 2026. [More code, less reuse: Investigating code quality and reviewer sentiment towards ai-generated pull requests](#). *Preprint*, arXiv:2601.21276.
- Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. 2024. [SWE-bench: Can language models resolve real-world github issues?](#) In *The Twelfth International Conference on Learning Representations*.
- Thomas Joshi, Shayan Chowdhury, and Fatih Uysal. 2025. [Swe-bench-cl: Continual learning for coding agents](#). *Preprint*, arXiv:2507.00014.
- Kabir Khandpur. 2025. [Swe-bench multilingual](#).
- Kimi. [Kimi k2.6](#).
- Hao Li, Haoxiang Zhang, and Ahmed E. Hassan. 2025. [The rise of ai teammates in software engineering \(se\) 3.0: How autonomous coding agents are reshaping software engineering](#). *Preprint*, arXiv:2507.15003.
- Simiao Liu, Fang Liu, Liehao Li, Xin Tan, Yinghao Zhu, Xiaoli Lian, and Li Zhang. 2025. [An empirical study on failures in automated issue solving](#). *Preprint*, arXiv:2509.13941.
- Yue Liu, Ratnadira Widyasari, Yanjie Zhao, Ivana Clairine Irsan, Junkai Chen, and David Lo. 2026. [Debt behind the ai boom: A large-scale empirical study of ai-generated code in the wild](#). *Preprint*, arXiv:2603.28592.
- MiniMax. [Minimax m2.5](#).
- OpenAI. [Gpt 5](#).
- OpenAI. 2026a. Codex: AI Coding Agent. <https://chatgpt.com/codex/>. Accessed: 2026-04-25.
- OpenAI. 2026b. Why swe-bench verified no longer measures frontier coding capabilities. <https://openai.com/index/why-we-no-longer-evaluate-swe-bench-verified/>.
- Gabriel Orlanski, Devjeet Roy, Alexander Yun, Changho Shin, Alex Gu, Albert Ge, Dyah Adila, Frederic Sala, and Aws Albarghouthi. 2026. [Slop-codebench: Benchmarking how coding agents degrade over long-horizon iterative tasks](#). *arXiv preprint arXiv:2603.24755*.

- Debalina Ghosh Paul, Hong Zhu, and Ian Bayley. 2025. [Investigating the smells of llm generated code](#). *Preprint*, arXiv:2510.03029.
- Ricardo Queirós. 2013. Codeskelgen-a program skeleton generator.
- Mehwish Riaz, Emilia Mendes, and Ewan Tempero. 2009. [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*, page 367–377, USA. IEEE Computer Society.
- Jungwoo Ryoo, Frederico Fonseca, and David S Janzen. 2008. 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*, pages 137–144. IEEE.
- Xin Sun, Daniel Ståhl, Kristian Sandahl, and Christoph Kessler. 2026. Quality assurance of llm-generated code: Addressing non-functional quality characteristics. *Journal of Systems and Software*, page 112885.
- Sizhe Wang, Zhengren Wang, Dongsheng Ma, Yongan Yu, Rui Ling, Zhiyu Li, Feiyu Xiong, and Wentao Zhang. 2026. [Codeflowbench: A multi-turn, iterative benchmark for complex code generation](#). *Preprint*, arXiv:2504.21751.
- Zhengren Wang, Rui Ling, Chufan Wang, Yongan Yu, Sizhe Wang, Zhiyu Li, Feiyu Xiong, and Wentao Zhang. 2025. [Maintaincoder: Maintainable code generation under dynamic requirements](#). *Preprint*, arXiv:2503.24260.
- Jingxuan Xu, Ken Deng, Weihao Li, Songwei Yu, Huaixi Tang, Haoyang Huang, Zhiyi Lai, Zizheng Zhan, Yanan Wu, Chenchen Zhang, Kepeng Lei, Yifan Yao, Xiping Lei, Wenqiang Zhu, Zongxian Feng, Han Li, Junqi Xiong, Dailin Li, Zuchen Gao, and 20 others. 2025. [Swe-compass: Towards unified evaluation of agentic coding abilities for large language models](#). *Preprint*, arXiv:2511.05459.
- John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik R Narasimhan, and Ofir Press. 2024. [SWE-agent: Agent-computer interfaces enable automated software engineering](#). In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*.
- Kai Yu, Zhenhao Zhou, Junhao Zeng, Ying Wang, Xueying Du, Zhiqiang Yuan, Junwei Liu, Ziyu Zhou, Yujia Wang, Chong Wang, and Xin Peng. 2026. [Does pass rate tell the whole story? evaluating design constraint compliance in llm-based issue resolution](#). *Preprint*, arXiv:2604.05955.
- Z.ai. [Glm 4.7](#).
- Aohan Zeng, Xin Lv, Qinkai Zheng, Zhenyu Hou, Bin Chen, Chengxing Xie, Cunxiang Wang, Da Yin, Hao Zeng, Jiajie Zhang, and 1 others. 2025. [Glm-4.5: Agentic, reasoning, and coding \(arc\) foundation models](#). *arXiv preprint arXiv:2508.06471*.
- Feng Zhang, Audris Mockus, Ying Zou, Foutse Khomh, and Ahmed E Hassan. 2013. How does context affect the distribution of software maintainability metrics? In *2013 IEEE International Conference on Software Maintenance*, pages 350–359. IEEE.

A Synthetic Problem Statement Construction for PR₁

Figure A1 shows the prompt used to generate synthetic PR₁ problem statements. Figure A2 shows the prompt template used for solving PR₁.

B Resolve Rate and Maintainability Metric Comparison on the Full Dataset

Table A1 shows the comparison of PR₂ resolution rates under HA and AA conditions.

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.

D LR Feature Details and LLM-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.

D.2 Patch Localization Features

The patch localization features measure where each chain’s PR₂ 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 PR₂ patch. We parse each post-patch source file with an AST parser (Python’s built-in `ast` module¹ 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 PR₂ patch with AA’s PR₂ 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 PR₁ 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 PR₁ diverges from Human PR₁. These divergences can silently propagate and cause failure during PR₂, 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 PR₂ 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 PR₁ Error Compounding into PR₂

The PR₁ behavioral drift judge in Appendix D.3 identifies instances where Agent PR₁ diverges from Human PR₁. A PR₁ divergence does not automatically mean the chain fails. Two scenarios can decouple it from the downstream failure. First, the agent’s PR₂ may rewrite the divergent function, replacing the drift with a different code block. Second, even when the drift survives, the benchmark’s unit tests may not exercise it. The question is therefore: did a PR₁ divergence persist into PR₂’s final

¹<https://docs.python.org/3/library/ast.html>

²<https://github.com/terryyin/lizard>

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: **PR₂ 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 PR₁ gate passed for both MiniMax and GLM-4.7.

Model	Provider	Reference
<i>Closed-source models</i>		
Claude 4.5 Sonnet	Anthropic	https://www.anthropic.com/claude
GPT-5	OpenAI	https://openai.com/index/introducing-gpt-5/
<i>Open-source models</i>		
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.

state, and did it directly relate to the failing test case? We use DeepSeek-V4 Pro to judge AA instances. Figure A5 and Figure A8 show the full system and user prompt with the output schema. We exclude instances labeled EQUIVALENT or STYLE by the PR₁ behavioral drift judge because they have no divergence to compound, which leaves 2,153 instances.

D.5 Human Audit of the PR₁ Fidelity Judge

We verify a stratified sample of the DeepSeek-V4 Pro judge’s outputs against the underlying code to assess the reliability of its IEC labels. We sampled 50 instances, with 56% labeled IEC and the remainder assigned non-IEC labels. Instances used as examples or illustrations elsewhere in the paper were excluded from the sample. Rare labels were over-sampled so that every label is represented. The sample spans all four models and all four benchmarks used in our evaluation. Samples range from

28 to 376 combined patch lines (median 109).

For each instance, the auditor was shown the judge’s label and reasoning together with the human and agent PR₁ implementations of the same stubbed target functions. The auditor then determined whether the assigned label was correct using the same criteria specified in the judge prompt. We audit the judge’s labels rather than obtain independent blind annotations given the difficulty of the task—assigning a label from scratch often requires repository-specific knowledge beyond the patch itself, such as whether an attribute may take a false value, and the sample of 50 instances covers 29 distinct repositories. As such, this procedure is a verification audit rather than an independent re-annotation, and we report confirmation rates.

The IEC/non-IEC classification underlying our analysis was confirmed in 48 of 50 instances (96.0%). Seven of these moved an item between two labels on the same side of the boundary, leaving

Library	Languages	CC	CogC	HV	LLOC
radon	Python	✓	—	✓	✓
complexity	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.

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

Did the PR₁ drift survive PR₂? PR₂ leaves the agent’s PR₁ drift unchanged on 85.9% of the 262 discordant instances, suggesting that the drift almost always persists into PR₂.

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 PR₁ drift directly causes the failing test on 20.6% of the 262 discordant instances. The remaining instances fail for undetermined reasons.

Table A4: **LLM-judge attribution on the HA-wins discordant cohort** (n=262 instances with a divergent agent PR₁; 30 PR₁-equivalent/style cases excluded). Drift survives PR₂ in 86% of cases; on 20.6% of the cohort the surviving drift directly traces to the failing test.

its IEC classification unchanged. The remaining two were reclassified from IEC to non-IEC, and no item was reclassified from non-IEC to IEC.

D.6 Robustness to Choice of LLM Judge

IEC is the only feature in our model that comes from an LLM judge. To check that the result does not depend on a judge model, we re-ran the PR₁ behavioral drift judge with Kimi K2.6 on the 405 discordant instances used to fit the model in Section 5. We rebuilt the IEC indicator from its labels and re-fit the same LR model by keeping everything constant. Table A6 compares the significant coefficients against Table 4; all three keep their sign, their significance, and their approximate magnitude.

D.7 Examples of IEC Divergence

Each figure below shows an instance with the human and agent implementations of the same function after PR₁ on the left, and what each becomes after the same follow-on issue is addressed on the right. Shaded lines mark the code identified as divergent. The five span all three divergence types and four languages: Figures A7 and A10 in C and Go, Figures A8 and A9 in Python, and Figure A11 in TypeScript.

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 PR₁ 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 `clef` and `cright` using `np.hstack`.

Figure A2: **Prompt template used for solving PR₁**. 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.

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., <code>dict.get(k, default)</code> , <code>int(x)</code> , <code>x</code> or <code>default</code>).
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., <code>None</code> vs. <code>[]</code> , scalar vs. tuple, different dict keys or layout).
EXTERNAL EFFECTS	Observable mutations or I/O differ (e.g., <code>stdout</code> or <code>log</code> 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, <code>raise NotImplementedError</code> , return paired with a <code>TODO</code> comment).

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

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 stubs 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 implemented", "TODO", "unimplemented", "finish later", "placeholder", or "fixme" * A return statement paired with a deferral comment (e.g., `return None # TODO`) A function that returns a constant value with no input-dependent logic is NOT automatically INCOMPLETE - it counts only if it also has a placeholder comment or stub wording. Otherwise it may be the intended implementation. The human reference never leaves stubs unfilled by construction, so this label is reserved for agent failures.

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 PR₁ 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 PR₁ 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 PR₁/PR₂ 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<sub>1</sub>/PR<sub>2</sub> compounding judge Defines the output schema.



Figure A7: **Example of a silent fallback.** In human PR<sub>1</sub>, clusterManagerMoveSlot returns zero on failure (line 4), propagating the failure to its caller. In agent PR<sub>1</sub> (lines 2–13), a failed call is handled differently: the code checks whether the error message contains certain keywords and, if so, prints the error and continues execution instead of returning zero. This behavior persists into agent PR<sub>2</sub> and causes the follow-up test to failure.

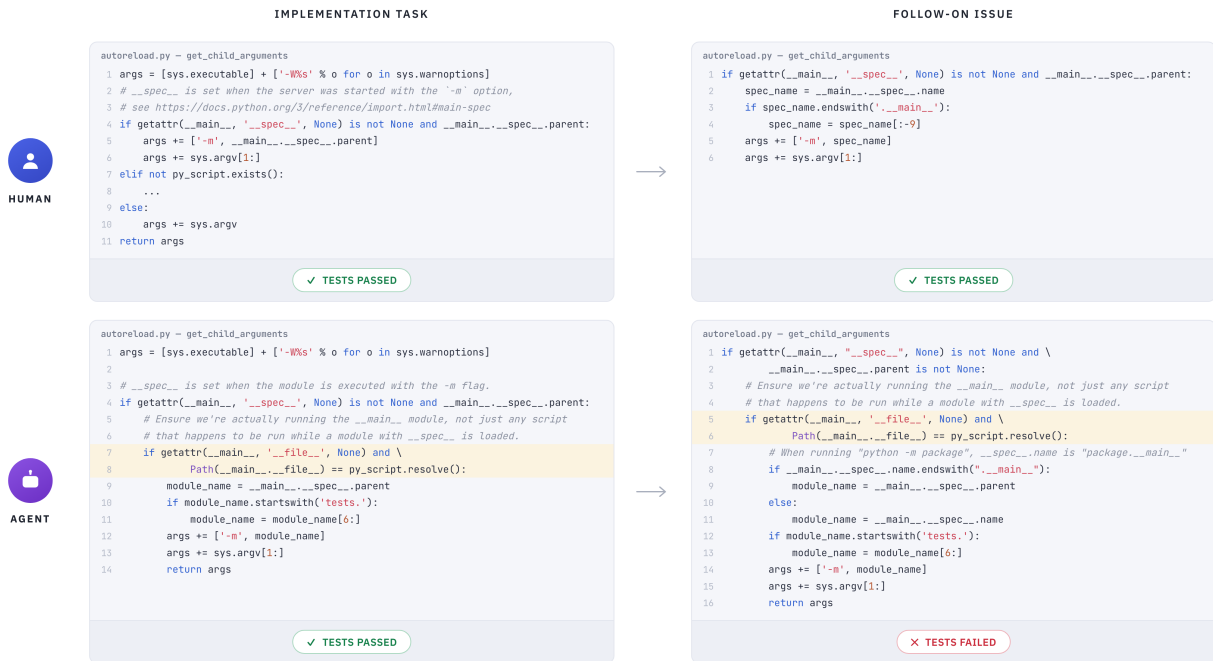


Figure A8: **Example of added input gating.** In Agent PR<sub>1</sub>, the condition on line 5 introduces an additional input check. This condition is carried over into Agent PR<sub>2</sub>, where it causes the follow-up test to fail.

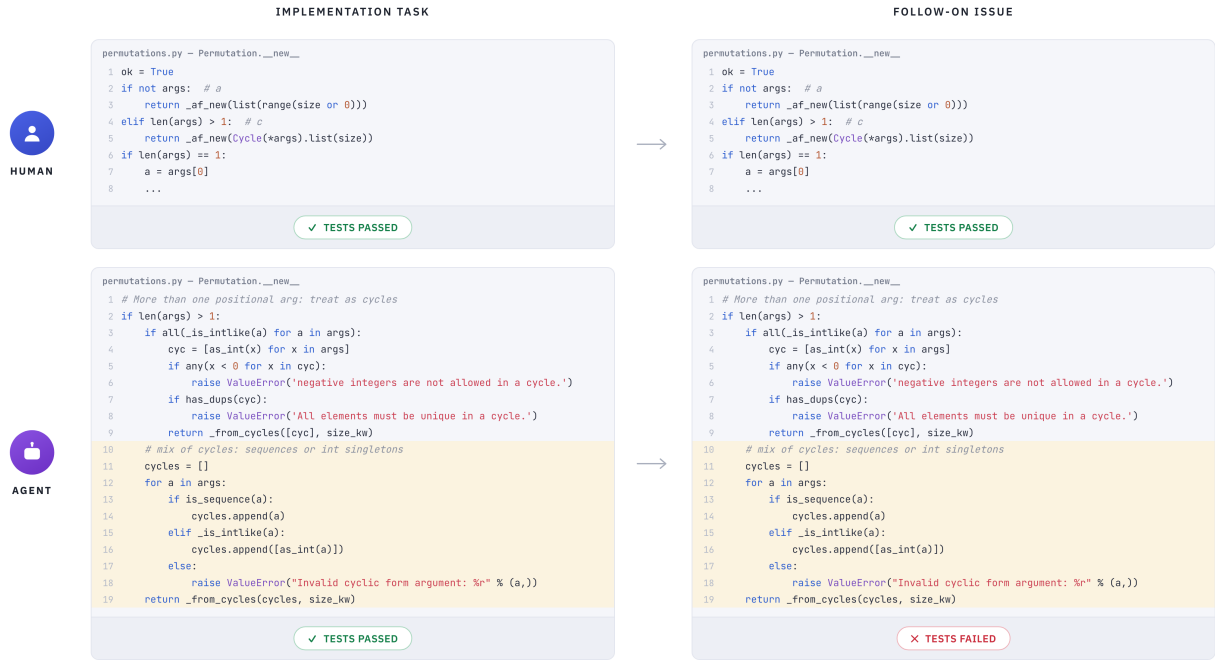


Figure A9: **Example of removed input gating.** The human authored PR<sub>1</sub> rejects a call made with two list arguments. The agent's PR<sub>1</sub> accepts them and builds a result instead (highlighted). PR<sub>2</sub> edits another function, leaving this behaviour exactly as PR<sub>2</sub> left it, so the test that expects the call to be rejected fails.

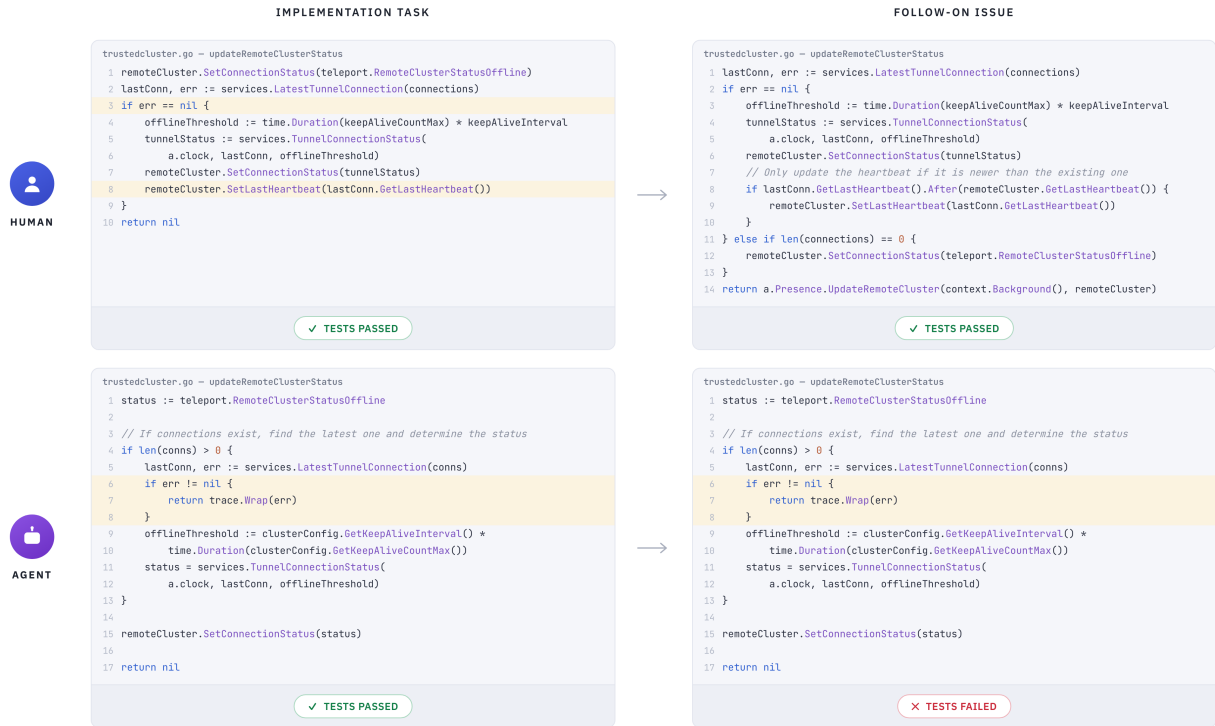


Figure A10: **Example of changed error handling.** When err is nil, human PR<sub>1</sub> records and stores the latest heartbeat. In agent PR<sub>1</sub>, the heartbeat is not stored; instead, the function returns to its caller. This behavior is preserved in Agent PR<sub>2</sub>, which returns an empty heartbeat date and causes the follow-up test to fail.

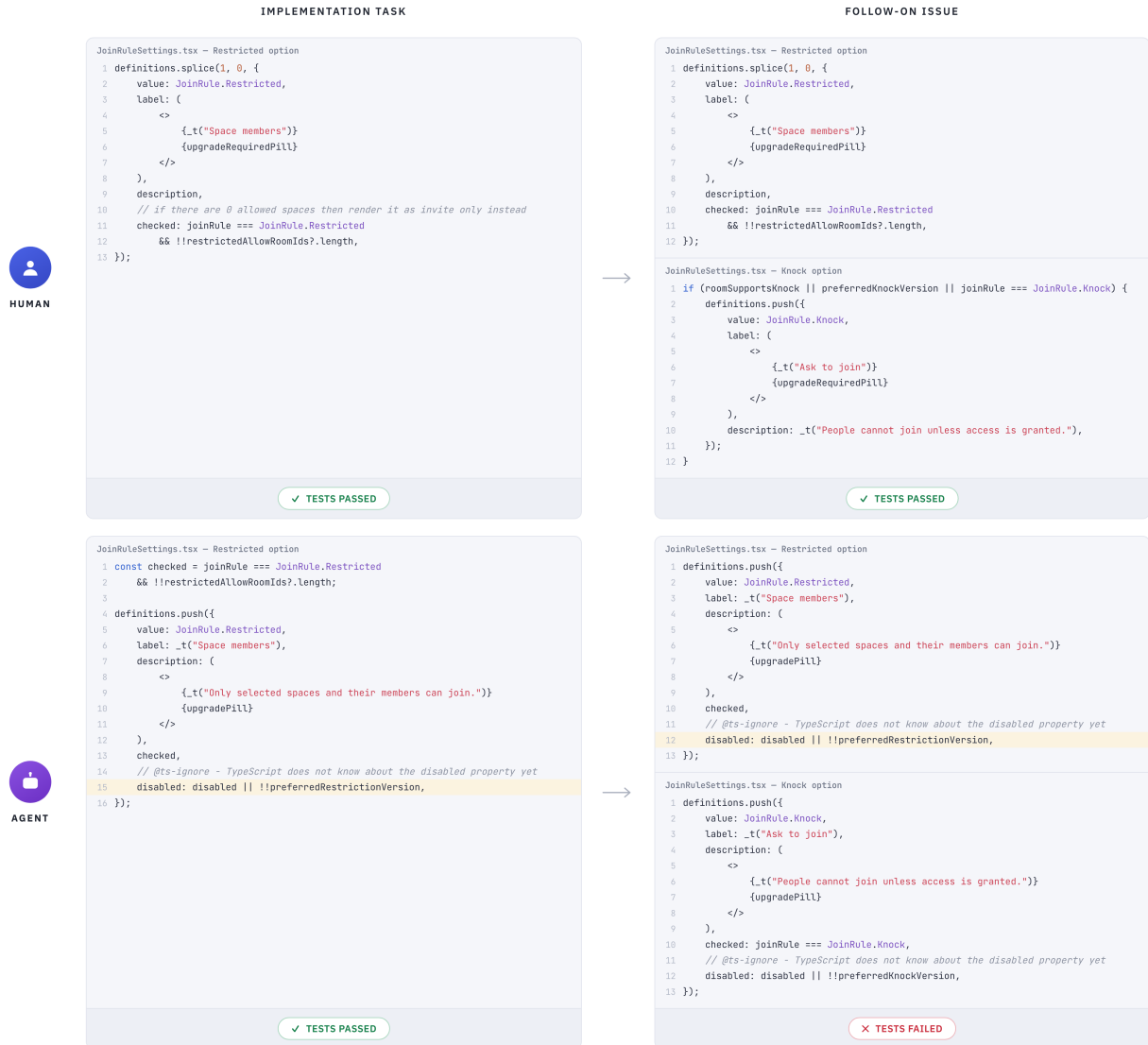


Figure A11: **Example of added input gating.** Human PR<sub>1</sub> does not include a disabled property, while Agent PR<sub>1</sub> adds one. This changes the frontend behavior: clicking the option no longer opens the dialog box. The same behavior persists in Agent PR<sub>2</sub> and is also applied to a newly added option. As a result, all nine follow-up tests fail because the expected dialog never appears.

Feature	DeepSeek V4 Pro		Kimi K2.6	
	OR	<i>p</i>	OR	<i>p</i>
$\Delta\text{LLOC}_{\text{PR}_2}$	1.88	0.042	1.83	0.042
IEC	1.83	0.011	1.70	0.039
Inst. resolve rate	1.42	0.003	1.42	0.003

Table A6: **Swapping the LLM judge does not change our findings.** We re-fit the model in Table 4 using IEC labels from a second judge, Kimi K2.6. The same three features stay significant, with similar odds ratios.