Coherence Collapse: Diagnosing Why Code Agents Fail After Reaching the Right Code

Anonymous ACL submission

Abstract

Code agents resolve 65–70% of SWE-bench Verified issues, but Pass@1 cannot tell us *why* the rest fail—and, as we show, capable-model failures are systematically misdiagnosed without trajectory data. We introduce TRAJEVAL, a training-free decomposition of agent trajectories into reference-patch-aligned search, read, and edit stages, and apply it across 16,758 trajectories spanning three architectures and seven models. The dominant failure of capable models is not localization: 60–69% of failures on SWE-Agent and OpenHands reach *and* edit the correct functions yet still produce incorrect patches, and the pattern persists for most models on the bash-only LiveSWEAgent. Within this Edit-Quality residual, we identify **Coherence Collapse**—the agent reaches correct code and then overwrites or thrashes it—as the largest theme, replicating across SWE-bench Verified and the multilingual PolyBench Verified. In 5 cases, the agent produces a patch **bit-identical to the gold reference** mid-trajectory and destroys it later; an edit-commit checkpoint recovers all 5 against the SWE-bench Docker harness. A reference-free consensus-driven variant yields a directional $+3.0$ pp Pass@1 measurement on GPT-5 (p=0.08).

1 Introduction

Code agents now resolve real GitHub issues at 65–70% on SWE-bench Verified (Jimenez et al., 2023; Wang et al., 2024; Xia et al., 2025b), yet the evaluation toolkit for them remains almost entirely outcome-based. Pass@1 treats each execution as a single binary outcome, mirroring the limitation that motivated process supervision for chain-of-thought reasoning (Wei et al., 2022; Huang et al., 2022; Schick et al., 2023): aggregate correctness conceals *where* and *why* reasoning succeeds or fails. Code is well-suited to a process-level shift because a correct patch is a precise, automatically extractable specification of which files, functions, and code regions the model needed to identify and modify; comparing an agent's trajectory against this reference converts unstructured tool-use traces into stage-specific signals over localization, comprehension, and modification.

We introduce TRAJEVAL, a trajectory-based diagnostic framework that decomposes agent execution into three stages (Figure 1): *Search* (file-level), *Read* (function-level inspection), and *Edit* (function-level modification). Each stage yields precision (efficiency) and recall (effectiveness) against the reference patch.

Applied at scale to 16,758 trajectories across three agent architectures and seven models, this lens surfaces a finding outcome metrics cannot expose: 60–69% of capable-model failures (GPT-5, GPT-5-mini, Qwen3-Coder-480B/30B, Qwen3-235B) on SWE-Agent and OpenHands reach *and* edit the correct functions yet still produce incorrect patches, and the pattern persists for the GPT-5 family and Qwen3-235B on the bash-only LiveSWEAgent (Section 4.1). Within this Edit-Quality residual, the dominant pattern is **Coherence Collapse**: the agent reaches correct code and then overwrites or thrashes it, replicating across SWE-bench Verified and the multilingual PolyBench Verified, with a length-independent sub-type that dissociates from context-window degradation (Section 4.2, Appendix A.23). The finding narrows the explanation for capable-model failures: localization alone cannot account for them. Reference-patch-aligned trajectory analysis is emerging concurrently (Ma et al., 2026a; Dihan and Khan, 2026); our contribution is to show that, at scale, it reveals an actionable failure mode—and that two interventions implied by the diagnosis (edit-commit checkpointing; parallel-sample consensus) admit direct end-to-end tests, reported in Section 4.3.

In summary, our contributions are:

- **A process-level lens for language-agent eval-**

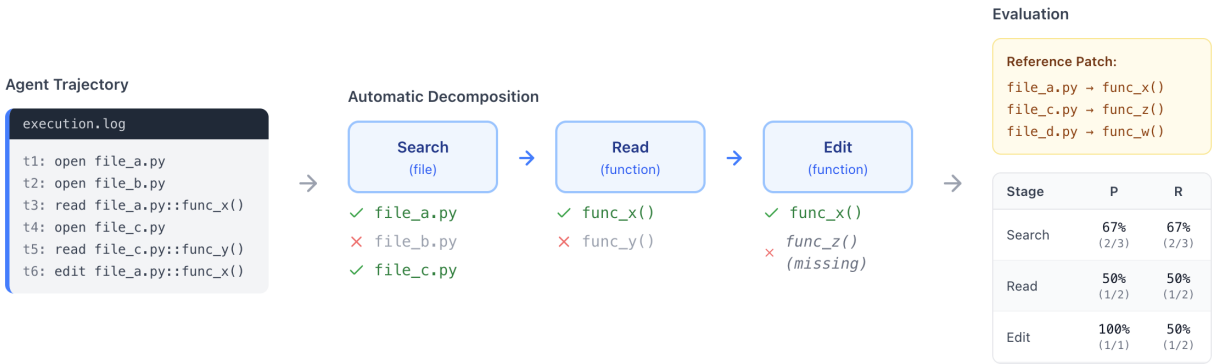

Figure 1: Overview of TRAJEVAL. Each agent execution trace is automatically decomposed into three stages—*search* (file-level), *read* (function-level), and *edit* (function-level)—and compared against the reference patch to yield stage-wise precision and recall.

uation on code. We introduce TRAJEVAL, a training-free, architecture-agnostic procedure that decomposes an agent trajectory into search, read, and edit stages and computes precision and recall against a reference patch, operationalizing process supervision for code agents using a verifiable reference signal complementary to outcome-based Pass@k. We apply it at scale: 16,758 trajectories across three architectures (SWE-Agent, OpenHands, LiveSWEAgent) and seven models from 8B to 480B parameters on SWE-bench Verified and PolyBench Verified.

- **Two findings with explicitly different scopes.** (i) *Stage-level:* Edit-Quality is the modal failure mode for capable models on SWE-Agent and OpenHands (60–69% of failures, all five capable models) and persists for the GPT-5 family and Qwen3-235B on the bash-only LiveSWEAgent (57–68%) (Section 4.1). (ii) *SWE-Agent only, sub-type taxonomy:* a 914-failure taxonomy identifies *Coherence Collapse* as the largest theme on both SWE-bench Verified (39.7%) and PolyBench Verified (32.3%, including Java 36.3% and JavaScript 42.1%); inter-annotator agreement on the headline 3-class distinction is substantial ($\kappa = 0.80$, Appendix A.19). Pass@1 progress for capable models is patch-quality-bound, not localization-bound.

- **Two interventions of differing evidence quality.** The diagnosis predicts that edit-commit checkpointing should recover Near-Correct Corrupted failures and that parallel-sample consensus should approximate the oracle's confirmatory signal without reference-patch access. The first is a clean *existence proof*: 5 intermediate edits bit-identical to gold pass the full SWE-bench Verified test suite when re-submitted to the Docker harness (Section 4.3). The second is a larger but *directional* measurement, sub-threshold at α=0.05: +3.0 pp Pass@1 on GPT-5 + SWE-Agent (p=0.08, n=500, Appendix A.22).

- **Reproducible artifact.** We release the trajectory-extraction pipeline, the per-trajectory feature CSV underlying every figure and table, and reproduction notebooks for each architecture (Anonymous, 2026).

2 TRAJEVAL: A Trajectory Analysis Lens

We formalize agent trajectories as stage-specific sets over files, functions, and edit targets, then compare them against a reference patch to compute stage-wise precision and recall (Figure 1). The procedure is lightweight, training-free, and architecture-agnostic, enabling consistent analysis across the three agent frameworks studied in Section 3.

2.1 Problem Formulation

Consider a code agent tasked with resolving an issue in repository $\mathcal{R}$ given an issue description d. The agent produces an execution trajectory $T = \{(a_t, o_t)\}_{t=1}^{n}$, where a_t denotes the action at step t and o_t the corresponding observation, and outputs a candidate patch $\hat{P}$. Given a reference patch P^*, our goal is to extract interpretable features from T that diagnose where agent behavior aligns with the reference solution.

The key insight is that P^* provides a canonical reference for the modified context of the task: which files were changed, which functions were involved, and which code regions were modified. We formalize this as the *golden context* $\mathcal{G} = (\mathcal{F}^*, \mathcal{H}^*)$, where $\mathcal{F}^*$ denotes the set of files modified by P^*, and $\mathcal{H}^*$ denotes the set of functions modified by P^*.

2.2 Three-Stage Decomposition

The three stages correspond to the three reference-patch projections the gold patch makes available—modified files, modified functions, and modified edit targets—yielding the coarsest decomposition that distinguishes navigation, comprehension, and modification without conflating them. Let $\mathcal{F}_T$ denote the set of files viewed during trajectory T, $\mathcal{H}_T$ denote the set of functions observed in agent observations, and $\mathcal{H}_{\hat{P}}$ denote the set of functions modified in the candidate patch.

Stage 1: Search (File-Level). Search quantifies the agent's ability to localize relevant files within the repository. Let P_s and R_s denote precision and recall for search:

$$P_s = \frac{|\mathcal{F}^* \cap \mathcal{F}_T|}{|\mathcal{F}_T|}, \quad R_s = \frac{|\mathcal{F}^* \cap \mathcal{F}_T|}{|\mathcal{F}^*|} \tag{1}$$

High recall indicates successful file discovery, while high precision reflects focused exploration without redundant navigation.

Stage 2: Read (Function-Level). File localization alone is insufficient: an agent may open a file without examining the relevant code region. Read quantifies function-level comprehension. Let P_r and R_r denote precision and recall for read:

$$P_r = \frac{|\mathcal{H}^* \cap \mathcal{H}_T|}{|\mathcal{H}_T|}, \quad R_r = \frac{|\mathcal{H}^* \cap \mathcal{H}_T|}{|\mathcal{H}^*|} \tag{2}$$

Read recall thus separates trajectories that examine the relevant function bodies from those that open a file but never inspect the buggy region.

Stage 3: Edit (Function-Level). Edit quantifies the localization accuracy of the agent's code modifications. Let P_e and R_e denote precision and recall for edit:

$$P_e = \frac{|\mathcal{H}^* \cap \mathcal{H}_{\hat{P}}|}{|\mathcal{H}_{\hat{P}}|}, \quad R_e = \frac{|\mathcal{H}^* \cap \mathcal{H}_{\hat{P}}|}{|\mathcal{H}^*|} \tag{3}$$

Edit metrics capture *localization* accuracy: an agent may achieve perfect edit recall while producing an incorrect patch. Semantic correctness is evaluated separately via Pass@k.

2.3 Golden Context

The golden context averages 1.4 files and 2.1 functions per instance. We refer to this strict reference as *Tier-0*; all precision and recall metrics in Sections 3–4 are computed against it. We deliberately use a tight reference rather than a structurally expanded one (e.g., one-hop transitive dependencies via the call graph) so that low precision unambiguously reflects exploration beyond what the patch required, rather than artifacts of the reference's permissiveness. Appendix A.13 reports a sensitivity analysis under a one-hop call-graph-expanded reference (*Tier-1*) that supports this design choice empirically.

Reading the reference patch as a projection. TRAJEVAL measures agent behavior under a reference-patch projection: an action is counted as "relevant" if and only if it intersects code that the gold solution touches. We adopt this framing intentionally rather than treating the reference as ground truth on what an agent "should" do. Two observations support its informativeness as a projection. First, among 350 instances solved by two or more models—a selection-biased subset of easier instances, since unsolved instances offer no successful patches to compare—100% show overlapping function-level modifications and 93.7% show *exact* function-level convergence (Section 4), indicating that on instances where comparison is possible, the modified-function set is largely invariant across successful agents rather than an arbitrary slice of the solution space. Second, the most actionable finding—Near-Correct Corrupted (Section 4.2)—is validated by direct diff-hunk comparison rather than by the precision/recall thresholds, so its identification does not depend on the reference being uniquely correct. Alternative valid fixes that modify disjoint locations nonetheless register as low-recall, a strict-by-design tradeoff we discuss in Limitations 6.

2.4 Feature Extraction

Given an agent-specific tool mapping $\phi : \mathcal{A} \to \{\texttt{view}, \texttt{edit}, \texttt{other}\}$, extraction iterates over the trajectory and accumulates $\mathcal{F}_T$, $\mathcal{H}_T$, $\mathcal{H}_{\hat{P}}$ before computing Eqs. 1–3 in $O(|T|)$ time. Function extraction is hybrid (tree-sitter parsing with regex fallback) and supports Python, Java, JavaScript, and TypeScript. The full procedure is given in Appendix A.10 (Algorithm 1).

What TRAJEVAL does not measure. TRAJEVAL captures *where* an agent operates, not whether the resulting code is correct. Two agents editing the same function with different patches receive identical edit metrics; debugging behaviors such as test interpretation, hypothesis formation, and post-edit verification appear only indirectly through their effect on stage-wise recall. The framework is designed to complement Pass@1, not replace it (Section 6).

3 Experiments

3.1 Experimental Setup

Models. We evaluate seven language models spanning 8B to 480B parameters: GPT-5, GPT-5-mini, Qwen3-8B, Qwen3-32B, Qwen3-235B, Qwen3-Coder-30B, and Qwen3-Coder-480B. The GPT-5 family is accessed via the OpenAI API; Qwen models via OpenRouter. All inference uses each provider's default decoding parameters; closed-API runs are not bit-deterministic, but our analysis pipeline uses fixed seed 42.

Benchmarks. **SWE-bench Verified** (Jimenez et al., 2023) contains 500 human-verified real-world GitHub issues from 12 Python repositories. **PolyBench Verified** (Rashid et al., 2025) contains 382 multilingual instances across Python, Java, JavaScript, and TypeScript from 20 repositories. An instance is resolved (Pass@1$=1$) if the agent's submitted patch passes all `FAIL_TO_PASS` tests without regressing any `PASS_TO_PASS` test, evaluated via the official SWE-bench Docker harness.

Agent architectures. We analyze three architectures that span the tool-primitive design spectrum, each run under its own default trajectory-budget and harness configuration: **SWE-Agent** (Yang et al., 2024) (specialized `view`/`str_replace`/bash), **OpenHands** (Wang et al., 2024) (`fsRead`/`fsWrite`/`executeBash`), and **LiveSWEAgent** (Xia et al., 2025b) (bash-only). This diversity lets us isolate model-intrinsic behavioral patterns from scaffold artifacts.

Trajectory counts. After excluding Qwen3-8B and Qwen3-32B from LiveSWEAgent due to persistent OpenRouter rate-limit/timeout errors with the bash-only harness (Appendix A.14), the design yields **9,500** SWE-bench and **7,258** PolyBench trajectories, **16,758** in total. The central capable-model finding is unaffected: both excluded models are capability-bound and fail at upstream Search regardless of scaffold (Section 4.1).

Code and analysis pipelines are available at an anonymous repository link (Anonymous, 2026).

3.2 Evaluation Protocol

We conduct three diagnostic analyses, each described in detail at first use:

- **Failure-mode partition** (Section 4.1): every failed trajectory is classified into one of four mutually exclusive stage-level buckets via cascading recall thresholds on R_s, R_r, R_e.
- **Edit-Quality taxonomy** (Section 4.2): the residual Edit-Quality bucket is decomposed via deletion-overlap, token-overlap, and edit-count signals; inter-annotator agreement is reported at three granularities (Appendix A.19).
- **Falsifiable predictions** (Section 4.3): two interventions implied by the diagnosis are tested end-to-end—edit-commit checkpointing (5 EXACT intermediate edits re-submitted to the SWE-bench Docker harness) and parallel-sample consensus (GPT-5 re-executed under SWE-Agent on all 500 SWE-bench Verified instances with confirmatory feedback driven by CONSENSUS$_3$ rather than the gold patch).

4 Results

We present three behavioral findings. Section 4.1 partitions every failed trajectory into mutually exclusive stage-level failure modes and shows that capable-model failures concentrate in the Edit-Quality bucket on SWE-Agent and OpenHands, with the pattern persisting for most capable models on the bash-only LiveSWEAgent. Section 4.2 decomposes that residual into a five-theme taxonomy on the SWE-Agent scaffold and identifies *Coherence Collapse* as the dominant pattern, replicated on PolyBench Verified. Section 4.3 states two predictions implied by the diagnosis and reports calibrated evidence for each: an existence proof for edit-commit checkpointing and a directional, sub-threshold measurement for parallel-sample consensus. Cross-agent behavioral patterns are reported in Appendix A.2.

4.1 Failure Mode Analysis

To make the per-model behavioral signatures concrete, we partition every failed trajectory (Pass@1

= 0) into one of four mutually exclusive, exhaustive stage-level failure modes using a cascading 0.5 threshold on stage-wise recall: **Search** ($R_s < 0.5$, never opens a sufficient share of relevant files), **Read** ($R_s \geq 0.5$, $R_r < 0.5$, reaches relevant files but misses relevant functions), **Edit-Target** ($R_r \geq 0.5$, $R_e < 0.5$, reaches relevant code but modifies wrong locations), and **Edit-Quality** ($R_e \geq 0.5$, reaches *and* edits the right locations yet produces an incorrect patch—a residual category no localization-stage signal can explain). Appendix A.12 re-buckets the same trajectories at $\tau \in \{0.3, 0.5, 0.7\}$ and shows that Edit-Quality remains the modal failure mode for all five capable models at every threshold (53–72%); the qualitative conclusions below are stable to this choice. The partition uses the strict Tier-0 reference (functions actually modified by the gold patch); under a one-hop call-graph-expanded Tier-1 reference, the absolute bucket distribution shifts mass toward Search while the model rank ordering by recall is preserved ($\rho = 1.00$). Tier-0 is therefore the tier under which downstream-failure discrimination is well-defined, and we report it accordingly (Appendix A.13).

Table 1 reports the resulting distribution for SWE-Agent on SWE-bench Verified (top) and PolyBench Verified (bottom); appendix tables 29–33 extend the same breakdown to the remaining four (scaffold, benchmark) cells.

Edit-Quality dominates capable models. Across the five GPT-5, Qwen3-Coder, and Qwen3-235B variants, 64–69% of failures fall in the Edit-Quality bucket: the agent reaches and edits the right functions, yet the generated patch is still incorrect. The pattern replicates across scaffolds: Edit-Quality is the modal failure mode for all five capable models on OpenHands (60.8–67.7%, Table 29) and for the GPT-5 family and Qwen3-235B on LiveSWEAgent (57.6–67.6%, Table 30); on the bash-only LiveSWEAgent harness, Qwen3-Coder-480B shifts to Read-dominant (40.3%) and Qwen3-Coder-30B becomes Search-modal (33.4%, narrowly above Edit-Quality at 33.2%), suggesting tool-interface effects compound the underlying diagnosis on bash-only scaffolds.

Capability-bound models fail upstream. Qwen3-8B's failures concentrate at Search (41.1%), and PolyBench amplifies this pattern further (Table 1, bottom): Search becomes the modal failure mode for every model except GPT-5, with the weighted Search rate rising from 19.3% on SWE-bench to 47.0% on PolyBench. Multilingual repositories appear to stress file discovery in particular, consistent with the per-tool reliability degradation reported in Section A.6.

Table 1: Failure-mode distribution among failed trajectories (Pass@1 = 0), SWE-Agent. Buckets are mutually exclusive and exhaustive; each row sums to 100%. n counts failed trajectories for which all six stage features could be extracted; trajectories where the agent terminated before producing parseable observations or edits are excluded, most strongly affecting weaker models.

Model	n	Search	Read	E.-Tgt.	E.-Qual.
SWE-bench Verified					
GPT-5	163	6.7	4.3	20.9	**68.1**
GPT-5-mini	162	8.6	8.0	14.2	**69.1**
Qwen3-Coder-480B	187	9.1	10.7	13.4	**66.8**
Qwen3-Coder-30B	231	11.7	10.8	10.8	**66.7**
Qwen3-235B	249	17.7	7.2	10.4	**64.7**
Qwen3-32B	362	25.7	9.4	27.3	**37.6**
Qwen3-8B	253	**41.1**	11.1	2.4	45.5
Weighted avg.	1607	19.3	9.0	14.8	**56.9**
PolyBench Verified					
GPT-5	208	18.8	**41.3**	8.7	31.2
GPT-5-mini	216	27.8	30.1	8.8	33.3
Qwen3-Coder-480B	261	28.0	**39.8**	11.1	21.1
Qwen3-Coder-30B	287	26.8	**36.6**	11.5	25.1
Qwen3-235B	295	**38.3**	25.8	11.2	24.7
Qwen3-32B	339	**61.4**	15.3	10.9	12.4
Qwen3-8B	353	**99.2**	0.6	0.3	0.0
Weighted avg.	1959	**47.0**	25.0	8.7	19.3

Distinct per-model signatures. Identical Pass@1=0 outcomes encode different failures across models (Edit-Target vs. Search), and tool-formatting friction compounds the picture: Qwen3-32B's 27.3% Edit-Target rate co-occurs with degraded file-manipulation tool reliability, and Qwen3-235B's 26.5% `submit` success rate is the most striking SWE-Agent toolkit anomaly (Appendix A.6).

4.2 Qualitative Taxonomy of Edit-Quality Failures

The Edit-Quality bucket (56.9% of all SWE-Agent + SWE-bench failures) is the dominant residual for every capable model but cannot be decomposed by stage-level features alone. We classify all 914 such failures into five themes comprising nine sub-types via a cascading-precedence rule on three trajectory signals (deletion overlap, addition-token overlap, gold-file edit count); the full rule and threshold-robustness check appear in Appendix A.18. We note upfront that Coherence Collapse and Semantic Error share a structural property—both reach

the correct location and produce incorrect code—and are operationally (not mechanistically) distinguished by token overlap with the gold edit. The Near-Correct Corrupted sub-type is more crisply distinguished, being validated by exact diff-hunk reconstruction (Appendix A.20); it is the most mechanistically grounded sub-claim in the taxonomy.

Table 2: Taxonomy of Edit-Quality failures (n=914, SWE-Agent on SWE-bench Verified). Sub-types are mutually exclusive (cascading precedence; Appendix A.18).

Theme	Sub-type	n	%
Coherence Collapse	Confused Thrashing	266	29.1
	Near-Correct Corrupted	97	10.6
Semantic Error	Wrong Branch Point	170	18.6
	Partial Understanding	67	7.3
	Wrong API Call	37	4.0
	Right Loc., Wrong Logic	28	3.1
Mis-loc.	Wrong Function in File	161	17.6
Scope	Incomplete Fix	64	7.0
	Missing Edge Case	8	0.9
Exec. Gap	Understood But No Edit	16	1.8
Total		**914**	**100.0**

Coherence Collapse (39.7% of the 914 Edit-Quality failures, i.e., 22.6% of all SWE-Agent failures). The agent reaches the correct code, often editing it, but its final patch abandons or breaks its own earlier progress. Two sub-types: *Confused Thrashing* (29.1%)—$\geq$ 3 edit attempts on the gold file, none persisting into the final diff; and *Near-Correct Corrupted* (10.6%)—an intermediate edit overlaps the gold patch ($> 50\%$ token overlap) and is subsequently overwritten or corrupted by a later edit. The two sub-types dissociate along trajectory length (Appendix A.23).

The Near-Correct Corrupted sub-type is the most actionable. An exact-hunk validation against the gold patch's diff structure (Table 3) shows that 30/97 (31%) reconstruct the gold patch's hunks at the EXACT or NEAR level, and 96/97 (99%) target the gold patch's code region rather than incidentally overlapping it. In the 5 EXACT cases, the agent produced a patch identical to the gold reference and then destroyed it later in the trajectory (Appendix A.24.3 walks through one such case in detail). These trajectories were not capability-bound: the model had already produced a correct fix, then deleted it.

Table 3: Diff-hunk validation of the 97 Near-Correct Corrupted trajectories. The 5 EXACT cases ground the existence proof in Section 4.3; 30 strong-match (EXACT + NEAR) bound the recovery ceiling.

Classification	Count	%
EXACT	5	5.2
NEAR	25	25.8
PARTIAL	66	68.0
DIFFERENT	1	1.0
Total	**97**	**100.0**

Coherence Collapse is not long-context degradation. The aggregate rate grows with length ($\rho = 0.32$, Q1 21.7% $\rightarrow$ Q4 63.7%), but sub-types dissociate sharply: Confused Thrashing drives this almost entirely ($\rho = 0.34$), while Near-Correct Corrupted is length-independent ($\rho = 0.001$, stable across quartiles). Longer trajectories produce more intervening steps before corruption but not greater probability of corruption (Appendix A.23).

Other themes and per-model patterns. *Semantic Error* (33.0%, right location but wrong fix logic) is stable across models (27–37%), suggesting task difficulty rather than capability. *Intra-File Mislocalization* (17.6%) edits the correct file but wrong function within it—a finer-grained failure than the inter-file Edit-Target bucket. *Scope/Completeness* (7.9%) peaks at 11–13% in GPT-5/GPT-5-mini, and *Execution Gap* (1.8%) peaks at 4.4% in Qwen3-32B. Per model (Appendix A.3): capable models (GPT-5/mini, 31–33% Coherence Collapse) maintain reasoning coherence at the cost of shallower fixes; Qwen3-32B (48.5%) loses the thread mid-fix most often.

Reliability. A second author independently labeled 100 SWE-bench Edit-Quality failures against the same cascading rule. Agreement on the headline 3-class distinction (Coherence Collapse / Semantic Error / Other) is substantial ($\kappa = 0.80$, 90% raw agreement); refinement to the 5-class theme level remains substantial ($\kappa = 0.71$). Disagreements concentrate on the Scope/Completeness boundary (15/23 disagreements) and never affect the 5 Near-Correct Corrupted EXACT cases (Appendix A.19).

Cross-benchmark replication on PolyBench. To test whether Coherence Collapse is a SWE-bench- or Python-specific artifact, we replicate the taxonomy on the multilingual PolyBench Verified (SWE-Agent, n=455 Edit-Quality failures; Ta-

ble 4). Coherence Collapse remains the largest single theme (32.3%); JavaScript (42.1%) exceeds the SWE-bench Python baseline (39.7%) and Java (36.3%) is comparable to it—narrowing the explanation: the pattern is not specific to Python or to the SWE-bench instance pool. The aggregate drop is driven by PolyBench Python (26.4%, the lowest cell), implicating benchmark composition rather than multilingual difficulty (full taxonomy and per-language tables in Appendix A.4).

Table 4: Cross-benchmark and per-language Coherence Collapse rate (% of Edit-Quality failures), SWE-Agent. JavaScript on PolyBench (42.1%) exceeds the SWE-bench Python baseline and Java (36.3%) is comparable, indicating the pattern is not Python- or benchmark-specific. Full taxonomy in Appendix A.4.

Benchmark	**Overall**	**Py**	**Java**	**JS**	**TS**
SWE-bench (n=914)	39.7	39.7	—	—	—
PolyBench (n=455)	32.3	26.4	36.3	**42.1**	32.1

4.3 Falsifiable Predictions from the Diagnosis

The diagnosis in Sections 4.1–4.2 implies specific predictions about the effect of interventions, each detectable from the trajectory alone. The two Edit-Quality sub-buckets imply different intervention classes—Coherence Collapse is an agent-loop problem (edit-commit checkpointing, trajectory summarization); Semantic Error is a code-generation-quality problem (test-driven refinement, multi-sample selection)—and we test the agent-loop predictions end-to-end below.

Prediction 1: Edit-commit checkpointing recovers the Near-Correct Corrupted residual. If an agent's intermediate edit passes the existing test suite, freezing it should prevent the corrupting edit that follows. Diff-hunk validation (Appendix A.20) shows that 30 of 97 NCC cases (31%) reconstruct gold-patch hunks at the EXACT or NEAR level, and 96/97 (99%) target the gold patch's code region. We re-evaluated the 5 EXACT intermediate edits against the SWE-bench test harness: *all 5 pass the full test suite*, establishing edit-commit checkpointing as a deployable existence proof rather than a hypothetical intervention. The realistic ceiling within NCC is bounded above by 30 (Appendix A.21); none of these signals require the reference patch at deployment.

Prediction 2: Parallel-sample consensus approximates the oracle's confirmatory signal. We define the *oracle* as a confirmatory feedback signal that, whenever the agent views a file present in the gold patch, injects a system message confirming that the file is gold-relevant; this fires on 490/500 instances and yields the $+4.6$ pp lift in Table 5. The oracle requires the reference patch at execution time and is therefore non-deployable. The 93.7% function-level convergence among successful agents (Appendix A.5) implies that when N independent samples agree on an edit location, that location is, with high probability, gold-aligned. For each SWE-bench instance, we construct CONSENSUS_N from files viewed by $\geq N$ of the 6 non-GPT-5 models (leave-one-out, no reference patch). At N=3, CONSENSUS_N recovers 97.3% (477/490) of the oracle's triggering cases.

We then measure consensus-driven feedback end-to-end: re-running GPT-5 on all 500 SWE-bench Verified instances under the same SWE-Agent harness with the confirmatory feedback trigger driven by CONSENSUS_3 rather than the gold patch (Table 5). The measurement is sub-threshold at $\alpha = 0.05$ ($+3.0$ pp Pass@1, 69.0% vs. 66.0%; McNemar's $p = 0.08$, n=500; $\sim$50% power for a $+3$ pp effect), but the effect direction matches the analytical $+4.5$ pp ceiling (under perfect-precision triggering) and the gain/loss decomposition (39 instances gained, 24 lost, net $+15$ instances) identifies a concrete mechanism for the residual gap.

That mechanism is the precision tradeoff: at CONSENSUS_3 precision of 52.4%, roughly half of triggered confirmations target non-gold files, occasionally misdirecting the agent—suggesting a higher-precision filter (N=4 or a learned re-ranker) as a direct improvement path. The chain-of-thought analogue is self-consistency decoding (Wang et al., 2023), here on tool-use trajectories rather than token sequences; Kim et al. (2026) use parallel sampling differently (patch ranking, not file-level confirmatory feedback).

Table 5: Consensus measurement on GPT-5, SWE-Agent + SWE-bench (n=500). Oracle uses gold patch; CONS_3 uses $\geq$ 3-of-6 non-GPT-5 model agreement (no reference-patch access).

Condition	**Pass@1**	Δ
Baseline	66.0% (330)	—
Oracle	70.6% (353)	+4.6 ($p < .001$)
CONS_3	69.0% (345)	+3.0 ($p = .08$)

5 Related Work

Positioning. Reference-patch-grounded trajectory analysis is concurrent work, not novel to this paper. Ma et al. (2026a) use the reference patch to score teacher trajectories on SWE-Gym for PRM training; Dihan and Khan (2026) train a lightweight PRM on SWE-bench. Our contribution is not the paradigm but its application as a training-free *evaluation* procedure, and—more centrally—the diagnoses and validated interventions it yields: *Coherence Collapse* as the dominant capable-model failure mode, 5 bit-identical-to-gold intermediate edits verified through the SWE-bench Docker harness, and a $+3.0$ pp end-to-end consensus measurement on GPT-5.

Process Supervision for Language Agents. Outcome-only metrics (Pass@k (Chen, 2021), reference-similarity (Ren et al., 2020)) discard trajectory information, motivating process reward models (Wei et al., 2022; Huang et al., 2022; Schick et al., 2023); TRAJEVAL is the evaluation analogue, complementary to PRM-style supervision (Ma et al., 2026a; Dihan and Khan, 2026) rather than replacing it.

Code Agents and SWE-bench Resolution. SWE-Agent (Yang et al., 2024), OpenHands (Wang et al., 2024), and LiveSWEAgent (Xia et al., 2025b) span the design spectrum from specialized commands to structured primitives to shell-only, letting us isolate model-intrinsic behavioral patterns from tool-interface artifacts. Agentless (Xia et al., 2025a) achieves competitive Pass@1 with a fixed three-phase pipeline (localization $\rightarrow$ repair $\rightarrow$ validation); our search/read/edit decomposition is reminiscent but applied *descriptively* to any agent's trajectory rather than prescriptively as architecture.

Trajectory and Behavior Analysis for Agents. General agent benchmarks (Liu et al., 2025; Zhou et al., 2024; Shi et al., 2017) and code-domain studies (Golnari et al., 2026) focus on outcomes rather than within-trajectory structure. A recent wave of code-agent trajectory studies overlaps more directly with our premise. Bouzenia and Pradel (2025) analyze thought-action-result patterns across RepairAgent/AutoCodeRover/OpenHands at the action-type level; TRAJEVAL adds reference-patch-aligned precision/recall to attribute behavior to specific gold-context targets. Majgaonkar et al. (2025) examine SWE-bench failures across three scaffolds via outcome statistics; we partition failures into a stage-level taxonomy and, distinctively, validate the most actionable sub-type by re-executing 5 intermediate edits through the SWE-bench Docker harness—confirming each is bit-identical to the gold reference and passes the full test suite (Section 4.2, Appendix A.21). Ma et al. (2026b) run 64,380 SWE-bench executions testing cross-framework rule transfer; our scope is orthogonal—a per-trajectory diagnostic rather than a rule-transfer study—and we operate on three scaffolds with seven models against the reference patch. Concurrent work by Sahoo et al. (2026) introduces "Lucky Pass" for trajectories that reach correct outcomes via incidental means; our *Coherence Collapse* captures the dual phenomenon—correct intermediate state subsequently lost—with stage-level Edit-Quality dominance shown on SWE-Agent and OpenHands (and persisting for most capable models on LiveSWEAgent) and 5 EXACT cases re-executed against the SWE-bench Docker harness as an existence proof.

6 Conclusion

TRAJEVAL—a reference-patch-aligned decomposition of agent trajectories into search, read, and edit stages—narrows the explanation for capable-model failures: localization alone cannot account for them. *Edit-Quality* is the modal failure mode on SWE-Agent and OpenHands (60–69%) and persists for the GPT-5 family and Qwen3-235B on LiveSWEAgent; within it, *Coherence Collapse*—the agent reaches correct code and then overwrites or thrashes it—is the dominant SWE-Agent pattern (39.7% on SWE-bench, 32.3% on PolyBench), with a length-independent Near-Correct Corrupted sub-type that is not reducible to context-window degradation. Two interventions admit direct tests: an edit-commit checkpoint recovers 5 bit-identical-to-gold intermediate edits against the SWE-bench Docker harness as an existence proof, and parallel-sample consensus yields a directional $+3.0$ pp Pass@1 on GPT-5 (p=0.08) without reference-patch access. Immediate follow-ups: replicating the taxonomy on OpenHands and LiveSWEAgent, and validating trajectory summarization for the larger Confused Thrashing sub-type. We release the trajectory-extraction pipeline, per-trajectory features, and checkpoint-validation artifacts.

Limitations

Reference-patch dependence. TRAJEVAL requires a reference patch and is therefore not applicable at deployment time, and alternative valid fixes modifying different code locations would register as low-recall. The latter is partially mitigated empirically: among 350 multiply-solved instances, 100% show overlapping function-level modifications and 93.7% show exact convergence (Section 4). A reference-tier sensitivity analysis (Appendix A.13) shows that Tier-1 expansion (11.7× larger) collapses the failure-mode partition's discrimination while preserving model rank ordering ($\rho = 1.00$).

Decomposition is localization-only and not exhaustive. The search/read/edit stages capture *where* an agent operates, not *whether the code is correct*: two agents editing the same function with different patches receive identical edit metrics, and behaviors central to debugging (bug reproduction, test interpretation, hypothesis formation, post-edit verification) appear only indirectly through their effect on stage-wise recall. TRAJEVAL is intended to complement Pass@1, not replace it.

Consensus measured on one (model, scaffold) cell. The end-to-end consensus measurement (+3.0 pp on GPT-5, $p = 0.08$) covers GPT-5 + SWE-Agent + SWE-bench Verified only; the same measurement on Qwen3-Coder-480B and on the OpenHands and LiveSWEAgent scaffolds is an immediate follow-up. The $p = 0.08$ result is below the analytical $+4.5$ pp ceiling (under perfect-precision triggering) and is sub-threshold at $\alpha = 0.05$; n=500 yields 80% power to detect a $+5$ pp effect and 50% power for $+3$ pp, so a larger benchmark or pooled measurement across scaffolds would tighten the confidence interval.

Edit-Quality taxonomy is heuristic-based, with multi-level IAA. The Section 4.2 taxonomy is computed from three trajectory signals (deletion overlap, addition-token overlap, gold-file edit count) under a cascading-precedence rule. Thresholds ($\delta = 0.5$, $\tau = 0.5$, $k = 3$) were selected via single-annotator inspection of a 50-trajectory calibration set, fixed before a second author independently labeled 100 SWE-bench Edit-Quality failures (Appendix A.19). Agreement on the headline 3-class distinction is substantial ($\kappa = 0.80$, 90% raw agreement), also substantial at 5-class refinement ($\kappa = 0.71$), and lower-substantial on the full 6-category rule ($\kappa = 0.68$). Disagreements concentrate on Scope/Completeness (15/23) and never affect the 5 Near-Correct Corrupted EXACT cases, which are validated independently by diff-hunk reconstruction.

Cross-scaffold taxonomy replication is untested. The Edit-Quality taxonomy is computed on SWE-Agent (SWE-bench and PolyBench) only; replication on OpenHands and LiveSWEAgent trajectories is left to future work.

Ethics Statement

This work analyzes execution trajectories from existing code agents on public benchmarks (SWE-bench Verified and PolyBench Verified) derived from open-source GitHub repositories under their original licenses. No human subjects, personal data, or private repositories were involved; all trajectories were produced by automated agents in sandboxed environments. The 16,758 trajectories were collected prior to this study; TRAJEVAL itself runs on CPU in minutes.

We see two potential risks. First, the oracle signals in Section 4 are non-deployable, and the end-to-end consensus measurement covers only a single (model, scaffold, benchmark) cell; we discourage interpreting either as guaranteed performance for end users. Second, stage-level diagnostics could in principle be used to game benchmark leaderboards by optimizing agents to match reference-patch trajectories rather than to solve underlying problems. We mitigate this by releasing the analysis pipeline as an evaluation tool rather than a training signal, and by explicitly distinguishing localization (where TRAJEVAL operates) from semantic correctness (which Pass@1 measures and TRAJEVAL does not replace).

References

Anonymous. 2026. Anonymous artifact for trajeval. https://anonymous.4open.science/r/TRAJEVAL/. Anonymous repository for double-blind review.

Islem Bouzenia and Michael Pradel. 2025. Understanding software engineering agents: A study of thought-action-result trajectories. *Preprint*, arXiv:2506.18824.

Mark Chen. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Dihan and Khan. 2026. Swe-shepherd: Advancing prms for reinforcing code agents. *Preprint*, arXiv:2604.10493.

Pareesa Ameneh Golnari, Adarsh Kumarappan, Wen Wen, Xiaoyu Liu, Gabriel Ryan, Yuting Sun, Shengyu Fu, and Elsie Nallipogu. 2026. Devbench: A realistic, developer-informed benchmark for code generation models. *Preprint*, arXiv:2601.11895.

Wenlong Huang, Pieter Abbeel, Deepak Pathak, and Igor Mordatch. 2022. Language models as zero-shot planners: Extracting actionable knowledge for embodied agents. In *Proceedings of the 39th International Conference on Machine Learning (ICML)*.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2023. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*.

Kim and 1 others. 2026. Scaling test-time compute for agentic coding. *Preprint*, arXiv:2604.16529.

Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, Shudan Zhang, Xiang Deng, Aohan Zeng, Zhengxiao Du, Chenhui Zhang, Sheng Shen, Tianjun Zhang, Yu Su, Huan Sun, and 3 others. 2025. Agentbench: Evaluating llms as agents. *Preprint*, arXiv:2308.03688.

Ma and 1 others. 2026a. From patches to trajectories: Privileged process supervision for swe agents. *Preprint*, arXiv:2605.21996.

Ma and 1 others. 2026b. Same signal, different semantics: Cross-framework behavioral analysis of swe agents. *Preprint*, arXiv:2605.18332.

Majgaonkar and 1 others. 2025. Understanding code agent behaviour: An empirical study of success and failure trajectories. *Preprint*, arXiv:2511.00197.

Muhammad Shihab Rashid, Christian Bock, Yuan Zhuang, Alexander Buchholz, Tim Esler, Simon Valentin, Luca Franceschi, Martin Wistuba, Prabhu Teja Sivaprasad, Woo Jung Kim, Anoop Deoras, Giovanni Zappella, and Laurent Callot. 2025. Swe-polybench: A multi-language benchmark for repository level evaluation of coding agents. *Preprint*, arXiv:2504.08703.

Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. Codebleu: a method for automatic evaluation of code synthesis. *Preprint*, arXiv:2009.10297.

Sahoo and 1 others. 2026. Agentlens: Revealing the lucky pass problem in swe-agent evaluation. *Preprint*, arXiv:2605.12925.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessí, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: language models can teach themselves to use tools. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23, Red Hook, NY, USA. Curran Associates Inc.

Tianlin Tim Shi, Andrej Karpathy, Linxi Jim Fan, Jonathan Hernandez, and Percy Liang. 2017. World of bits: an open-domain platform for web-based agents. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ICML'17, page 3135–3144. JMLR.org.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, and 1 others. 2024. Openhands: An open platform for ai software developers as generalist agents. *arXiv preprint arXiv:2407.16741*.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2023. Self-consistency improves chain of thought reasoning in language models. In *International Conference on Learning Representations*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. 2022. Chain-of-thought prompting elicits reasoning in large language models. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, NIPS '22, Red Hook, NY, USA. Curran Associates Inc.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2025a. Demystifying llm-based software engineering agents. *Proc. ACM Softw. Eng.*, 2(FSE).

Chunqiu Steven Xia, Zhe Wang, Yan Yang, Yuxiang Wei, and Lingming Zhang. 2025b. Live-swe-agent: Can software engineering agents self-evolve on the fly? *arXiv preprint arXiv:2511.13646*.

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652.

Shuyan Zhou, Frank F. Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, Uri Alon, and Graham Neubig. 2024. Webarena: A realistic web environment for building autonomous agents. *Preprint*, arXiv:2307.13854.

A Appendix

A.1 Appendix Overview

This appendix provides supplementary results supporting the main paper findings:

- Cross-agent behavioral summary table and over-exploration metrics (Appendix A.2)
- Cross-benchmark Edit-Quality taxonomy on PolyBench Verified, including per-language slice (Appendix A.4)
- Robustness to alternative correct solutions: full convergence analysis (Appendix A.5)
- Complete efficiency metrics, token consumption, and tool success rates on SWE-bench and PolyBench
- Trajectory feature extraction methodology
- Edit-Quality taxonomy assignment rule and threshold robustness (Appendix A.18)
- Inter-annotator agreement on the cascading rule, reported at three levels of granularity (Appendix A.19)
- Reference-tier sensitivity (Tier-0 vs. Tier-1; Appendix A.13)
- Failure-mode breakdowns for the remaining (scaffold, benchmark) cells
- Near-Correct Corrupted diff-hunk validation (Appendix A.20)
- Edit-commit checkpoint validation: 5/5 EXACT cases pass test suite (Appendix A.21)
- Consensus-as-relevance-signal validation (Appendix A.22)
- Trajectory length and Coherence Collapse dissociation (Appendix A.23)
- Edit-Quality worked examples (Appendix A.24)

A.2 Cross-Agent Behavioral Patterns: Full Tables

Table 6 reports the complete per-agent and per-model behavioral metrics referenced in Section 4.

Search recall averages 0.72–0.78 across agents, indicating that locating the correct files is a shared challenge regardless of tool implementation. Read recall shows more variance (0.58–0.71), with OpenHands achieving highest values, plausibly due to its structured file-reading interface. Edit recall ranges from 0.45–0.68, with SWE-Agent performing best, likely due to its specialized `str_replace` command. Precision is uniformly low across stages and agents, indicating substantial over-exploration: agents inspect 8–12× more files and 17–25× more functions than necessary. The efficiency table also reveals deployment-relevant operational anomalies, most notably Qwen3-235B's 26.5% submit success rate despite near-perfect view reliability; detailed tool-level analysis follows in Table 12.

Metric	SWE-Agent	OpenHands	LiveSWE
Search Recall	0.78	0.74	0.72
Read Recall	0.65	0.71	0.58
Edit Recall	0.68	0.52	0.45
Search Precision	0.12	0.08	0.10
Read Precision	0.05	0.06	0.04
Edit Precision	0.25	0.18	0.15
Avg. Tool Calls	30.7	24.2	13.5
Files Viewed	8.2	9.5	7.8

(a) Cross-Agent Behavioral Summary.

Model	Pass@1	Tools/Task	Submit %	View %
GPT-5	66.0%	30.7	99.9%	99.7%
GPT-5-mini	60.4%	32.2	99.9%	99.9%
Qwen3-Coder-480B	61.2%	50.3	99.9%	99.7%
Qwen3-Coder-30B	52.6%	50.5	100%	99.8%
Qwen3-235B	45.0%	42.5	**26.5%**	99.9%
Qwen3-8B	13.2%	48.6	93.2%	99.2%
Qwen3-32B	14.8%	43.8	99.3%	100%

(b) Efficiency Metrics.

Table 6: Behavioral patterns and efficiency metrics on SWE-bench Verified.

A.3 Edit-Quality Themes by Model

Table 7: Edit-Quality failure themes by model (% of each model's Edit-Quality failures). Coh. = Coherence Collapse; Sem. = Semantic Error; Loc. = Intra-File Mislocalization; Scope = Scope/Completeness; Gap = Execution Gap.

Model	Coh.	Sem.	Loc.	Scope	Gap
GPT-5	33.3	34.2	20.7	10.8	0.9
GPT-5-mini	31.2	32.1	22.3	13.4	0.9
Qwen3-Coder-480B	44.0	32.0	15.2	7.2	1.6
Qwen3-Coder-30B	38.3	36.4	17.5	6.5	1.3
Qwen3-235B	39.1	37.3	14.9	6.8	1.9
Qwen3-32B	**48.5**	27.2	14.7	5.1	**4.4**
Qwen3-8B	41.7	30.4	20.0	7.0	0.9

A.4 Cross-Benchmark Taxonomy: PolyBench Verified

This section reports the full Edit-Quality taxonomy applied to PolyBench Verified, replicating the procedure used for SWE-bench Verified in Section 4.2. The same cascading-precedence rule (Appendix A.18) is applied to all 455 SWE-Agent + PolyBench Edit-Quality failures (the cell-specific count from Table 31's Edit-Quality column).

Table 8: Edit-Quality theme distribution: SWE-bench Verified vs. PolyBench Verified (SWE-Agent scaffold). Coherence Collapse remains the largest single theme on both benchmarks; Semantic Error and Intra-File Mislocalization are within ± 1 pp across benchmarks. The cross-benchmark drop in Coherence Collapse is driven by PolyBench Python (Table 9) rather than by a multilingual effect.

Theme	**SWE-bench** (n=914)	**PolyBench** (n=455)	Δ (pp)
Coherence Collapse	39.7%	32.3%	-7.4
Semantic Error	33.0%	33.9%	$+0.9$
Intra-File Mislocalization	17.6%	16.9%	-0.7
Scope/Completeness	7.9%	8.3%	$+0.4$
Execution Gap	1.8%	8.6%	$+6.8$

Per-language slice. Table 9 reports the same theme distribution stratified by programming language within PolyBench Verified.

Table 9: PolyBench Edit-Quality taxonomy by language (SWE-Agent). JavaScript (42.1%) exceeds and Java (36.3%) is comparable to the SWE-bench Python baseline (39.7%), contradicting the "monolingual vs. multilingual" hypothesis. The aggregate cross-benchmark drop is concentrated in PolyBench Python (26.4%).

Language	n	**Coh.**	**Sem.**	**Loc.**	**Scope**	**Gap**
Java	135	36.3	36.3	17.8	4.4	5.2
JavaScript	76	**42.1**	39.5	11.8	3.9	2.6
Python	216	26.4	31.0	18.1	12.5	**12.0**
TypeScript	28	32.1	28.6	17.9	7.1	**14.3**
All	455	32.3	33.9	16.9	8.3	8.6

Three observations. *Coherence Collapse generalizes across languages and benchmarks.* JavaScript (42.1%) exceeds and Java (36.3%) is comparable to the 39.7% SWE-bench Python baseline. Combined with the SWE-bench result, the phenomenon appears at $\geq 32\%$ in five independent (language, benchmark) cells, providing empirical evidence against the most natural alternative—that the finding reflects a Python or SWE-bench artifact.

The cross-benchmark drop is benchmark-composition, not multilingual. The 7.4 pp aggregate drop in Coherence Collapse on PolyBench is concentrated in *PolyBench Python* (26.4%, 13.3 pp below SWE-bench Python). Java and JavaScript actually exhibit higher Coherence Collapse than the SWE-bench baseline. PolyBench Python issues differ in repository selection, issue type, and test structure from SWE-bench; these benchmark-composition factors—not language—drive the aggregate difference.

Execution Gap rises sharply on PolyBench, with two distinct mechanisms. The Execution Gap theme (agent reads the gold file but never attempts an edit on it) jumps from 1.8% to 8.6%, concentrated in PolyBench Python (12.0%) and TypeScript (14.3%). We manually inspected 10 of the 16 PolyBench Python Execution Gap instances and identified two underlying mechanisms: (i) *code paralysis* (4/10): the agent views the gold file 3–5 times, produces only reproduction scripts, and submits an empty or unrelated patch—concentrated on hard instances (`django-13513` recurs across 4 models); (ii) *fix misattribution* (6/10): the agent reads the gold file thoroughly (mean 5.2 views) but concludes the fix belongs in an adjacent module—for example, on a `QuerySet.bulk_create` bug in `query.py`, the agent edits `postgresql/operations.py` instead. Both mechanisms confirm the rise is not a parser artifact: Execution Gap is about reading the right code and failing to act on it, either through paralysis on hard instances or through misattribution to a related but incorrect module. This is consistent with PolyBench being curated to be challenging.

Scope. Like the SWE-bench taxonomy in Section 4.2, the PolyBench taxonomy is computed only on the SWE-Agent scaffold; whether the cross-language pattern holds for OpenHands and LiveSWEAgent remains untested.

A.5 Robustness to Alternative Correct Solutions

To determine whether the reference patch represents the unique correct solution or whether alternative valid fixes may modify different code locations, we analyze 350 instances that were successfully resolved by two or more models. At the function level, 100% of multiply-resolved instances show overlapping modifications—all models edit at least one common function. 93.7% of instances

show *exact* function-level convergence, where all successful models modify precisely the same functions. At the file level, 99.4% of instances show at least partial overlap, with only 0.6% modifying completely disjoint file sets. This convergence suggests that SWE-bench issues have substantial convergence in modified locations, and—as we show in Section 4.3—this convergence is exploitable: a CONSENSUS$_3$ relevance signal built from $\geq$ 3-of-6 model agreement on viewed files yields a measured +3.0 pp Pass@1 lift on GPT-5 without reference-patch access (Appendix A.22).

We observe 106 resolved instances (8.6% of successful resolutions) with low edit recall (<0.8). Manual inspection of 10 cases with edit recall = 0 reveals two patterns: (1) *partial completions* where agents modify a subset of required functions (e.g., editing 2 of 4 golden locations), and (2) *alternative solutions* where agents fix the bug via completely different code paths (e.g., editing `contenttypes/fields.py` when the reference patch modifies `fields/__init__.py`). Both patterns successfully resolve the issue by passing all tests.

A.6 Complete Model Results: SWE-bench Verified

A.6.1 Efficiency Metrics

Table 10 presents complete efficiency metrics for all seven models evaluated on SWE-bench Verified. GPT-5 family demonstrates a clear efficiency gradient: GPT-5 achieves the lowest average tool usage at 30.71 calls per task. Qwen3-235B achieves comparable efficiency to GPT-5-mini despite lower bash usage (15.25 vs 16.84), while Qwen3-Coder-30B and Qwen3-Coder-480B show elevated tool counts (50.45 and 50.30) with moderate localization percentages (20.5% and 23.4%), indicating more deliberate but verbose problem-solving strategies.

A.6.2 Token Consumption by Tool

Table 11 presents complete token consumption statistics for all tools across all models on SWE-bench Verified. Bash output tokens exhibit the highest variance across models. The view operation shows consistent input requirements (25.7–29.0 tokens) but substantial output variance (448–1682 tokens), indicating models inspect similar file regions but receive different context lengths based on file structure. The str_replace operation reveals model-specific editing patterns: Qwen models require higher input tokens (190–616) compared to GPT models (232–327), suggesting more verbose edit specifications, while output tokens remain consistent (96–457), reflecting similar confirmation message lengths.

A.6.3 Tool Success Rates

Table 12 presents complete tool success rates for all models on SWE-bench Verified. The most striking anomaly is Qwen3-235B's submit success rate of only 26.5% across 3557 attempts, compared to 99.9% for GPT-5 and 100% for Qwen3-Coder-30B, indicating systematic issues with patch formatting or validation logic in this configuration. Qwen3-32B also shows degraded performance across several file-manipulation operations: `create` drops to 78.9%, `str_replace` to 92.6%, and `undo_edit` to 71.7%. In contrast, `view` maintains near-perfect reliability (>99%) across all models, confirming file inspection as the most robust tool in the SWE-Agent toolkit. These operational failures are not captured by the stage-wise trajectory features but remain important for practical deployment.

A.7 Complete Model Results: PolyBench Verified

A.7.1 Efficiency Metrics

Table 13 presents complete efficiency metrics for all seven models evaluated on PolyBench Verified. Cross-benchmark comparison reveals that multilingual tasks increase complexity: GPT-5 requires 25% more tool invocations on PolyBench (38.38) compared to SWE-bench (30.71). The Qwen3-8B anomaly stands out with unusually low tool usage (24.89) and minimal localization (4.1%), suggesting early termination or limited exploration capability rather than efficient problem-solving. Qwen3-32B exhibits the highest localization percentage (71.6%), indicating extensive code navigation before editing—a pattern that correlates with its degraded tool success rates. Pipe usage increases substantially on PolyBench, with Qwen3-32B reaching 0.64 pipes per bash call, reflecting the need for complex command chaining in multilingual codebases with diverse build systems and directory structures.

A.7.2 Token Consumption by Tool

Table 14 presents complete token consumption statistics for all tools across all models on PolyBench Verified. Bash output tokens increase dramatically on PolyBench: GPT-5-mini reaches 21460 tokens (vs 1497 on SWE-bench), a 14× in-

Table 10: Complete Efficiency Metrics (SWE-bench Verified). Columns: μ_{tool} mean tool calls per trajectory; μ_{LLM} mean LLM invocations per trajectory; μ_{bash} mean bash invocations per trajectory; P_{pipe} mean pipe (|) operators per bash command (a count, not a probability; values > 1 indicate multi-pipe chains); **Loc.** total localization-class commands across all trajectories; **Loc.%** fraction of bash commands classified as localization. Qwen3-8B was run on SWE-Agent + SWE-bench (n=500 trajectories, included in the failure-mode partition Table 1) but is excluded from per-tool aggregate tables due to its shorter average trajectories (7–10 tool calls vs. 20–40 for other models), which yield insufficient per-tool-type samples for reliable aggregation.

Agent	μ_{tool}	μ_{LLM}	μ_{bash}	P_{pipe}	**Loc.**	**Loc.%**
GPT-5	30.71	30.71	15.36	0.22	2558	33.3
GPT-5-mini	32.16	32.16	16.84	0.24	3493	41.5
Qwen3-235B	42.53	42.53	15.25	0.20	1682	22.1
Qwen3-Coder-30B	50.45	50.45	27.44	0.42	2815	20.5
Qwen3-32B	48.59	48.59	13.62	0.23	2469	36.2
Qwen3-Coder-480B	50.30	50.30	24.80	0.36	2900	23.4

Table 11: Complete Token Consumption (SWE-bench Verified). Qwen3-8B excluded from per-tool aggregates; see Table 10 caption.

Tool		**GPT-5**	**GPT-5-mini**	**Q-235B**	**Q-30B**	**Q-32B**	**Q-480B**
bash	In	76.0	96.8	41.8	87.0	44.7	54.2
	Out	5210	1497	906	315	1301	593
view	In	25.7	27.1	27.4	26.6	29.0	27.2
	Out	1682	1406	902	776	448	715
str_replace	In	322	327	560	616	190	555
	Out	316	379	418	457	132	432
create	In	253	228	258	721	235	570
	Out	15	16	18	15	20	16
insert	In	149	111	132	61	103	—
	Out	231	244	261	182	234	—
undo_edit	In	17.0	17.3	17.8	—	20.4	17.1
	Out	14.3	15.0	15.0	—	15.7	14.4
submit	In	5.3	5.3	5.3	5.3	5.3	5.3
	Out	1864	351	2407	2583	614	1318

Table 12: Complete Tool Success Rates (SWE-bench Verified). Qwen3-8B excluded from per-tool aggregates; see Table 10 caption.

Tool	**GPT-5**	**GPT-5-mini**	**Q-235B**	**Q-30B**	**Q-32B**	**Q-480B**
bash	96.2%	96.4%	96.3%	97.3%	94.5%	97.7%
	(7213)	(7953)	(7149)	(13225)	(6417)	(11905)
view	99.7%	99.9%	99.9%	99.8%	99.2%	99.7%
	(4549)	(4200)	(4843)	(5970)	(7521)	(6165)
str_replace	98.8%	98.9%	97.2%	99.5%	92.6%	98.4%
	(1467)	(1656)	(4665)	(2086)	(8137)	(2748)
create	98.8%	98.9%	92.7%	99.3%	78.9%	97.2%
	(659)	(264)	(1035)	(2901)	(1816)	(2493)
insert	100%	100%	100%	100%	92.6%	—
	(94)	(23)	(12)	(1)	(122)	
undo_edit	100%	100%	100%	—	71.7%	100%
	(67)	(15)	(1)		(46)	(2)
submit	99.9%	99.9%	26.5%	100%	93.2%	99.9%
	(1305)	(1897)	(3557)	(1042)	(236)	(1838)

Table 13: Complete Efficiency Metrics (PolyBench Verified)

Agent	μ_{tool}	μ_{LLM}	μ_{bash}	P_{pipe}	**Loc.**	**Loc.%**
GPT-5	38.38	38.38	18.06	0.34	2204	35.6
GPT-5-mini	34.62	34.62	18.89	0.37	2447	44.4
Qwen3-235B	43.12	43.12	18.26	0.41	2081	32.3
Qwen3-32B	48.45	48.45	30.26	0.64	7644	71.6
Qwen3-8B	24.89	24.89	24.89	0.02	362	4.1
Qwen3-Coder-30B	46.05	46.05	23.66	0.57	2149	25.7
Qwen3-Coder-480B	50.21	50.21	24.53	0.53	2435	28.1

crease reflecting verbose compilation errors and test output in Java/TypeScript projects. The str_replace output tokens for GPT-5 reach 1396 on PolyBench versus 316 on SWE-bench, a 4.4× increase indicating more complex edit contexts in multilingual codebases. File creation (create) requires substantially more input tokens on PolyBench (283–647 tokens) compared to SWE-bench (228–721 tokens), reflecting boilerplate requirements and import statements in statically-typed languages. The insert operation shows extreme variance for GPT-5-mini (2537 output tokens), suggesting verbose feedback when inserting code into Java/TypeScript files with complex type annotations.

A.7.3 Tool Success Rates

Table 15 presents complete tool success rates for all models on PolyBench Verified. Cross-benchmark degradation is evident: Qwen3-32B's create success drops from 78.9% (SWE-bench) to 58.7% (PolyBench), and str_replace falls from 92.6% to 83.5%. The undo_edit operation shows severe degradation for Qwen3-32B, dropping from 71.7% to 43.8%, indicating that edit state management becomes increasingly problematic in multilingual contexts. Qwen3-235B's submit success improves from 26.5% (SWE-bench) to 65.3% (PolyBench), though this remains substantially below other models, suggesting partial but incomplete resolution of patch formatting issues. Qwen3-8B shows complete absence of editor tool usage (all marked as —), confirming that this model fails to engage with file manipulation operations and relies exclusively on bash commands. The GPT-5 family maintains robust performance across benchmarks, with view operations achieving 100% success and str_replace remaining above 98.5%.

A.8 OpenHands Results

A.8.1 SWE-bench Verified

Table 16 presents efficiency metrics for OpenHands on SWE-bench Verified. OpenHands demonstrates distinct behavioral patterns compared to SWE-agent: lower average tool usage (13.98–45.71 vs 30.71–50.45) but higher localization command percentages, with Qwen3-32B reaching 97.6% localization. The Qwen3-Coder-480B configuration requires the most tool invocations (45.71) while maintaining moderate localization (61.7%), suggesting a more balanced exploration-exploitation strategy. Pipe usage varies substantially, with Qwen3-235B showing anomalously high usage (1.10 pipes per bash call), indicating command chaining patterns with multiple pipes per invocation.

Table 17 presents token consumption for OpenHands tools on SWE-bench. The fsWrite operation shows substantial variance in input tokens (230–694), reflecting different code generation verbosity across models. Bash output tokens range from 195 (Qwen3-Coder-480B) to 4503 (Qwen3-32B), a 23× difference indicating divergent command strategies.

Table 18 presents tool success rates for OpenHands on SWE-bench. The fsWrite operation exhibits the most variance, ranging from 29.0% (Qwen3-8B) to 97.7% (Qwen3-Coder-480B). This 3.4× difference highlights model-specific challenges with file creation and modification. The fsRead operation shows degraded performance for GPT-5-mini (63.2%) and Qwen3-32B (59.8%), suggesting file path resolution issues. System and think tools maintain 100% success across all models, confirming their reliability as non-destructive operations.

A.8.2 PolyBench Verified

Table 19 presents efficiency metrics for OpenHands on PolyBench Verified. Cross-benchmark comparison reveals increased complexity: GPT-5 requires 20% more tool invocations on PolyBench (29.03) compared to SWE-bench (24.18). Qwen3-Coder-480B maintains the highest tool usage (47.45) with moderate localization (66.2%). The Qwen3-Coder-30B configuration shows elevated pipe usage (0.88 pipes per bash call), reflecting complex command chaining requirements in multilingual codebases.

Table 20 presents token consumption for OpenHands on PolyBench. Bash output tokens increase substantially: GPT-5 reaches 2329 tokens (vs 1046 on SWE-bench), while Qwen3-8B shows extreme output (6628 tokens), indicating verbose error messages in multilingual compilation. The fsWrite input tokens remain relatively stable across benchmarks, suggesting consistent code generation patterns regardless of target language.

Table 21 presents tool success rates for OpenHands on PolyBench. The fsWrite operation shows degraded performance across most models: GPT-5 drops from 89.0% (SWE-bench) to 66.7% (PolyBench), and Qwen3-8B falls from 29.0% to 20.6%. This cross-benchmark degradation reflects increased complexity in multilingual file manipulation. The fsRead operation also degrades, with

Table 14: Complete Token Consumption (PolyBench Verified)

Tool		GPT-5	GPT-5-mini	Q-235B	Q-32B	Q-8B	Q-30B	Q-480B
bash	In	88.7	102.0	45.0	64.0	51.7	53.5	49.2
	Out	12297	21460	3983	4344	968	2472	1539
view	In	26.8	30.1	32.1	30.8	—	28.3	29.0
	Out	1800	1501	1012	711	—	1244	1068
str_replace	In	306	314	438	190	—	597	565
	Out	1396	1313	611	130	—	544	555
create	In	336	341	293	283	—	647	625
	Out	62	97	38	16	—	143	189
insert	In	161	220	84	67	—	352	—
	Out	510	2537	161	197	—	518	—
undo_edit	In	19.3	17.0	—	19.7	—	—	—
	Out	16.7	14.2	—	13.9	—	—	—
submit	In	5.3	5.3	5.3	5.3	—	5.3	5.3
	Out	3699	2882	6411	8258	—	4961	12060

Table 15: Complete Tool Success Rates (PolyBench Verified)

Tool	GPT-5	GPT-5-mini	Q-235B	Q-32B	Q-8B	Q-30B	Q-480B
bash	96.2%	95.9%	94.5%	92.9%	82.8%	95.9%	96.3%
	(5921)	(5266)	(6152)	(10514)	(8786)	(8043)	(8368)
view	100%	99.9%	99.8%	96.6%	—	99.9%	99.9%
	(4239)	(2846)	(4177)	(2993)		(4676)	(5391)
str_replace	99.4%	99.8%	98.2%	83.5%	—	99.9%	99.2%
	(1757)	(1177)	(2946)	(3032)		(1381)	(1957)
create	99.1%	99.6%	92.4%	58.7%	—	99.8%	98.2%
	(450)	(233)	(1106)	(387)		(1545)	(1448)
insert	100%	100%	100%	96.9%	—	100%	—
	(109)	(19)	(38)	(64)		(1)	
undo_edit	100%	100%	—	43.8%	—	—	—
	(44)	(20)		(16)			
submit	99.8%	98.1%	65.3%	94.7%	—	100%	100%
	(646)	(521)	(801)	(95)		(610)	(561)

Table 16: OpenHands Efficiency Metrics (SWE-bench Verified)

Agent	μ_{tool}	μ_{LLM}	μ_{bash}	P_{pipe}	Loc.	Loc.%
GPT-5	24.18	24.18	18.84	0.56	7048	74.8
GPT-5-mini	20.06	20.06	15.15	0.49	5627	74.3
Qwen3-235B	26.26	26.26	9.65	1.10	3650	75.7
Qwen3-32B	13.98	13.98	4.49	0.42	2192	97.6
Qwen3-8B	26.55	26.55	8.97	0.45	4188	93.4
Qwen3-Coder-480B	45.71	45.71	20.61	0.74	6354	61.7
Qwen3-Coder-30B	37.48	37.48	18.92	0.76	5579	59.0

Table 17: OpenHands Token Consumption (SWE-bench Verified)

Tool		GPT-5	GPT-5-mini	Q-235B	Q-32B	Q-8B	Q-480B	Q-30B
executeBash	In	31.6	34.3	28.1	33.0	22.9	31.6	41.0
	Out	1046	1555	549	4503	1962	195	208
fsRead	In	18.7	17.4	18.4	18.7	18.7	18.2	18.5
	Out	1694	1610	1305	510	500	695	753
fsWrite	In	277	417	324	230	374	593	694
	Out	332	250	267	141	131	206	244
system	In	3104	3104	3528	3528	3528	3528	3528
	Out	5.8	5.8	5.8	5.8	5.8	5.8	5.8
think	In	481	296	252	537	391	479	634
	Out	7.3	7.3	7.3	7.3	7.3	7.3	7.3

Table 18: OpenHands Tool Success Rates (SWE-bench Verified)

Tool	GPT-5	GPT-5-mini	Q-235B	Q-32B	Q-8B	Q-480B	Q-30B
executeBash	95.4%	96.3%	91.5%	98.0%	97.0%	94.9%	95.1%
	(9418)	(7575)	(4823)	(2246)	(4483)	(10304)	(9462)
fsRead	87.4%	63.2%	98.2%	59.8%	95.5%	97.8%	98.2%
	(952)	(960)	(2953)	(3260)	(4159)	(7930)	(5345)
fsWrite	89.0%	74.1%	76.1%	44.5%	29.0%	97.7%	95.6%
	(1168)	(953)	(927)	(483)	(1143)	(2803)	(1817)
system	100%	100%	100%	100%	100%	100%	100%
	(498)	(492)	(500)	(500)	(500)	(500)	(500)
think	100%	100%	100%	100%	100%	100%	100%
	(55)	(20)	(3929)	(501)	(1964)	(1317)	(1618)

Table 19: OpenHands Efficiency Metrics (PolyBench Verified)

Agent	μ_{tool}	μ_{LLM}	μ_{bash}	P_{pipe}	Loc.	Loc.%
GPT-5	29.03	29.03	20.18	0.56	5884	76.3
GPT-5-mini	25.45	25.45	19.59	0.51	5449	72.8
Qwen3-235B	30.40	30.40	14.68	0.84	4511	80.5
Qwen3-32B	18.03	18.03	8.83	0.93	2786	82.6
Qwen3-8B	25.43	25.43	10.98	0.30	3463	82.6
Qwen3-Coder-480B	47.45	47.45	19.58	0.66	4956	66.2
Qwen3-Coder-30B	37.80	37.80	17.38	0.88	4421	66.6

Table 20: OpenHands Token Consumption (PolyBench Verified)

Tool		GPT-5	GPT-5-mini	Q-235B	Q-32B	Q-8B	Q-480B	Q-30B
executeBash	In	40.3	38.6	30.8	41.5	26.5	26.0	35.3
	Out	2329	3086	1016	1290	6628	524	623
fsRead	In	23.5	24.4	24.2	23.9	25.0	23.0	23.1
	Out	1516	1122	1348	575	498	690	823
fsWrite	In	311	383	363	279	472	572	606
	Out	308	253	240	154	124	270	285
system	In	3104	3104	3528	3528	3528	3528	3528
	Out	5.8	5.8	5.8	5.8	5.8	5.8	5.8
think	In	688	349	281	138	455	523	701
	Out	7.3	7.3	7.3	7.4	7.3	7.3	7.3

GPT-5-mini dropping from 63.2% to 61.0%.

A.9 LiveSWEAgent Results

A.9.1 SWE-bench Verified

Table 22 presents efficiency metrics for LiveSWEAgent on SWE-bench Verified. LiveSWEAgent operates exclusively through bash commands, with tool usage equal to bash usage across all models. The GPT-5 family shows efficient operation (13.45–14.67 calls per task), while Qwen models require substantially more invocations (42.38–50.56). Localization percentages range from 50.9% (Qwen3-Coder-30B) to 66.4% (Qwen3-Coder-480B), indicating moderate exploration overhead. Pipe usage is notably higher for GPT-5 models (0.74–0.75 pipes per bash call) compared to Qwen3-235B (0.26), suggesting different command composition strategies.

Table 23 presents token consumption for LiveSWEAgent on SWE-bench. Bash input tokens range from 79.8 (Qwen3-Coder-30B) to 158.8 (GPT-5), reflecting different command verbosity. Output tokens show extreme variance: Qwen3-235B generates 14688 tokens per command on average, compared to 152 tokens for Qwen3-Coder-480B—a 97× difference indicating fundamentally different execution patterns, likely due to verbose error handling or extensive test output.

Table 24 presents tool success rates for LiveSWEAgent on SWE-bench. Bash success rates range from 92.9% (Qwen3-Coder-30B) to 98.2% (Qwen3-Coder-480B). The GPT-5 family maintains consistent performance (94.3–95.8%), while Qwen models show more variance. The high success rates across all models confirm bash command execution as a reliable operation, though the variance suggests model-specific command formatting issues.

A.9.2 PolyBench Verified

Table 25 presents efficiency metrics for LiveSWEAgent on PolyBench Verified. Cross-benchmark comparison shows mixed patterns: GPT-5 requires slightly more invocations on PolyBench (15.23 vs 13.45), while Qwen3-235B shows reduced usage (25.10 vs 43.76). Localization percentages increase for most models on PolyBench, with Qwen3-Coder-480B reaching 68.9%, reflecting additional navigation requirements in multilingual codebases. Pipe usage increases substantially for GPT-5 (0.89 vs 0.75 pipes per bash call), indicating more complex command chaining for multilingual tasks.

Table 26 presents token consumption for LiveSWEAgent on PolyBench. Bash input tokens remain relatively stable across benchmarks, with GPT-5 at 148.8 tokens (vs 158.8 on SWE-bench). Output tokens decrease substantially for Qwen3-235B (230 vs 14688), suggesting more efficient command execution in multilingual contexts, possibly due to different error handling patterns or reduced test verbosity.

Table 27 presents tool success rates for LiveSWEAgent on PolyBench. Success rates remain high across all models (93.3–97.9%), with Qwen3-Coder-30B achieving the highest rate (97.9%) despite moderate efficiency. GPT-5-mini shows slight degradation (93.3% vs 95.8% on SWE-bench), while Qwen3-Coder-480B maintains consistent performance (97.6% vs 98.2%). These

Table 21: OpenHands Tool Success Rates (PolyBench Verified)

Tool	GPT-5	GPT-5-mini	Q-235B	Q-32B	Q-8B	Q-480B	Q-30B
executeBash	98.2%	97.8%	96.2%	98.6%	98.1%	95.3%	96.0%
	(7710)	(7483)	(5607)	(3374)	(4195)	(7481)	(6641)
fsRead	89.2%	61.0%	97.3%	66.3%	90.0%	93.2%	97.7%
	(1217)	(677)	(2883)	(2401)	(2465)	(7016)	(4621)
fsWrite	66.7%	55.4%	54.5%	39.8%	20.6%	90.5%	77.8%
	(1722)	(1140)	(1264)	(530)	(880)	(2087)	(1540)
system	100%	100%	100%	100%	100%	100%	100%
	(365)	(375)	(382)	(382)	(380)	(382)	(382)
think	100%	100%	100%	100%	100%	100%	100%
	(75)	(47)	(1478)	(200)	(824)	(1158)	(1255)

Table 22: LiveSWEAgent Efficiency Metrics (SWE-bench Verified). Qwen3-8B and Qwen3-32B excluded due to OpenRouter API errors (Section 3, coverage note).

Agent	μ_{tool}	μ_{LLM}	μ_{bash}	P_{pipe}	Loc.	Loc.%
GPT-5	13.45	13.45	13.45	0.75	4224	62.8
GPT-5-mini	14.67	14.67	14.67	0.74	4545	61.9
Qwen3-235B	43.76	43.76	43.76	0.26	14447	66.0
Qwen3-Coder-30B	42.38	42.38	42.38	0.44	10782	50.9
Qwen3-Coder-480B	50.56	50.56	50.56	0.21	16793	66.4

Table 23: LiveSWEAgent Token Consumption (SWE-bench Verified). Qwen3-8B and Qwen3-32B excluded due to OpenRouter API errors.

Tool		GPT-5	GPT-5-mini	Q-235B	Q-30B	Q-480B
executeBash	In	158.8	96.7	122.3	79.8	97.5
	Out	528	430	14688	817	152

Table 24: LiveSWEAgent Tool Success Rates (SWE-bench Verified). Qwen3-8B and Qwen3-32B excluded due to OpenRouter API errors.

Tool	GPT-5	GPT-5-mini	Q-235B	Q-30B	Q-480B
executeBash	94.3%	95.8%	97.8%	92.9%	98.2%
	(5434)	(6010)	(20880)	(17701)	(24474)

Table 25: LiveSWEAgent Efficiency Metrics (PolyBench Verified). Qwen3-8B and Qwen3-32B excluded due to OpenRouter API errors.

Agent	μ_{tool}	μ_{LLM}	μ_{bash}	P_{pipe}	Loc.	Loc.%
GPT-5	15.23	15.23	15.23	0.89	3906	67.1
GPT-5-mini	14.28	14.28	14.28	0.86	3674	67.3
Qwen3-235B	25.10	25.10	25.10	0.31	6299	65.7
Qwen3-Coder-30B	42.33	42.33	42.33	0.46	7703	47.6
Qwen3-Coder-480B	46.78	46.78	46.78	0.26	12311	68.9

Table 26: LiveSWEAgent Token Consumption (PolyBench Verified). Qwen3-8B and Qwen3-32B excluded due to OpenRouter API errors.

Tool		GPT-5	GPT-5-mini	Q-235B	Q-30B	Q-480B
executeBash	In	148.8	95.5	143.6	89.3	101.1
	Out	501	466	230	131	191

results confirm bash-only operation as a robust strategy across benchmark types.

A.10 Algorithm: Trajectory Feature Extraction

Algorithm 1 Trajectory Feature Extraction

Require: Trajectory $T = \{(a_t, o_t)\}_{t=1}^{n}$, Reference patch P^*, Tool mapping ϕ
Ensure: Feature vector $\mathbf{m} \in \mathbb{R}^6$

```
F*, H* ← ParsePatch(P*)
F_T, H_T, H_P̂ ← ∅, ∅, ∅
for (a_t, o_t) ∈ T do
    if φ(a_t) = view then
        F_T ← F_T ∪ ExtractFiles(a_t)
        H_T ← H_T ∪ ExtractFunctions(o_t)
    else if φ(a_t) = edit then
        H_P̂ ← H_P̂ ∪ ExtractEditTargets(a_t)
    end if
end for
m ← [P_s, R_s, P_r, R_r, P_e, R_e] using Eqs. 1–3
Return m
```

A.11 Trajectory Feature Extraction Methodology

This section details the feature extraction pipeline used to compute the six trajectory metrics.

A.11.1 Golden Context Definition

We define **golden context** as the minimal code elements required to generate the correct patch:

1. **Golden Files:** Files modified in the ground-truth patch, extracted from diff headers (`diff -git a/... b/...`)
2. **Golden Functions:** Functions containing modified lines, extracted from hunk headers (`@@ ... @@ function_name`)
3. **Golden Lines:** Line ranges modified in each file, extracted from hunk markers

A.11.2 Agent Action Extraction

For each agent framework, we extract actions from trajectory logs:

SWE-Agent Actions:

- `view`: File opened for inspection → files_viewed
- `str_replace`: Edit operation → files_edited
- `bash`: Commands like `cat`, `grep` → files_viewed

OpenHands Actions:

- `fsRead`: File read operation → files_viewed
- `fsWrite`: File write operation → files_edited
- `executeBash`: Shell commands parsed for file references

LiveSWEAgent Actions:

- `executeBash`: All operations via bash commands
- Parse `cat`, `head`, `vim`, `sed` for files_viewed
- Parse `patch`, `sed -i`, file redirects for files_edited

A.11.3 Metric Computation

Given golden context G and agent actions A:

$$\text{Search Precision} = \frac{|G_{\text{files}} \cap A_{\text{viewed}}|}{|A_{\text{viewed}}|} \quad (4)$$

$$\text{Search Recall} = \frac{|G_{\text{files}} \cap A_{\text{viewed}}|}{|G_{\text{files}}|} \quad (5)$$

$$\text{Read Precision} = \frac{|G_{\text{funcs}} \cap A_{\text{funcs_read}}|}{|A_{\text{funcs_read}}|} \quad (6)$$

$$\text{Read Recall} = \frac{|G_{\text{funcs}} \cap A_{\text{funcs_read}}|}{|G_{\text{funcs}}|} \quad (7)$$

$$\text{Edit Precision} = \frac{|G_{\text{funcs}} \cap A_{\text{funcs_edited}}|}{|A_{\text{funcs_edited}}|} \quad (8)$$

$$\text{Edit Recall} = \frac{|G_{\text{funcs}} \cap A_{\text{funcs_edited}}|}{|G_{\text{funcs}}|} \quad (9)$$

Table 27: LiveSWEAgent Tool Success Rates (PolyBench Verified). Qwen3-8B and Qwen3-32B excluded due to OpenRouter API errors.

Tool	GPT-5	GPT-5-mini	Q-235B	Q-30B	Q-480B
executeBash	94.5% (4653)	93.3% (4235)	95.3% (8657)	97.9% (14358)	97.6% (17184)

A.12 Threshold Sensitivity for Failure-Mode Buckets

The failure-mode partition in Section 4.1 uses a cascading 0.5 threshold on stage-wise recall. We re-bucket the same SWE-Agent + SWE-bench failures at $\tau \in \{0.3, 0.5, 0.7\}$ to test whether the qualitative findings depend on this choice. Lower τ relaxes the bar for "reaching" a stage; higher τ makes it stricter.

Table 28: Failure-mode distribution under varying recall thresholds τ, SWE-Agent on SWE-bench Verified. Each row sums to 100%. Bold entries mark the modal bucket per row. Edit-Quality remains the modal bucket for all five capable models at every threshold (53–72%); Qwen3-32B and Qwen3-8B move between Search-modal and Edit-Quality-modal but stay in the upstream-failure regime (Search + Read $\geq$ 50% at $\tau = 0.7$).

Model	τ	n	Search	Read	E.-Tgt.	E.-Qual.
GPT-5	0.3	163	5.5	4.9	19.6	**69.9**
	0.5	163	6.7	4.3	20.9	**68.1**
	0.7	163	18.4	4.9	20.9	**55.8**
GPT-5-mini	0.3	162	6.8	7.4	14.2	**71.6**
	0.5	162	8.6	8.0	14.2	**69.1**
	0.7	162	16.7	8.6	16.0	**58.6**
Qwen3-Coder-480B	0.3	187	7.0	7.5	15.0	**70.6**
	0.5	187	9.1	10.7	13.4	**66.8**
	0.7	187	18.7	15.0	10.7	**55.6**
Qwen3-Coder-30B	0.3	231	10.0	8.2	11.7	**70.1**
	0.5	231	11.7	10.8	10.8	**66.7**
	0.7	231	22.1	11.7	11.3	**55.0**
Qwen3-235B	0.3	249	15.7	6.0	10.8	**67.5**
	0.5	249	17.7	7.2	10.4	**64.7**
	0.7	249	**28.1**	6.0	12.4	53.4
Qwen3-32B	0.3	362	24.6	9.9	27.1	**38.4**
	0.5	362	25.7	9.4	**27.3**	37.6
	0.7	362	**32.0**	9.4	24.6	34.0
Qwen3-8B	0.3	253	39.1	9.9	2.4	**48.6**
	0.5	253	41.1	11.1	2.4	**45.5**
	0.7	253	**47.8**	9.1	2.8	40.3
Weighted avg.		1607				
at $\tau = 0.3$			17.6	8.0	15.0	**59.4**
at $\tau = 0.5$			19.3	9.0	14.8	**56.9**
at $\tau = 0.7$			28.0	9.3	14.5	**48.2**

Findings. Three observations support the robustness of the Section 4.1 claims. (1) Edit-Quality remains the modal failure mode for all five capable models (GPT-5, GPT-5-mini, Qwen3-Coder-480B/30B, Qwen3-235B) at every threshold; the rate spans 53–72%, well above any other bucket. (2) The capable-vs-capability-bound split is preserved: Qwen3-8B and Qwen3-32B remain in the upstream-failure regime (Search + Read $\geq$ 50% at τ=0.7), while GPT-5-family models stay Edit-Quality-dominated. (3) The shift at τ=0.7 is concentrated in Search at the expense of Edit-Quality (capable models lose 10–14 points of Edit-Quality, gain $\sim$10 points of Search), consistent with stricter bars demoting "barely reached" trajectories from downstream to upstream buckets—an effect on labels rather than on the underlying behavior. The qualitative conclusions of Section 4.1 are stable.

A.13 Reference-Tier Sensitivity: Tier-0 vs. Tier-1 Golden Context

The golden context $\mathcal{G}$ used throughout the paper (Section 2) is the strict set of files and functions modified by the reference patch (*Tier-0*). A natural alternative is to expand the reference one hop along the call graph—the patched function plus its direct callers and callees and their containing files (*Tier-1*)—on the grounds that an agent that reads a caller of the buggy function is doing useful work even if the caller is not itself patched. We re-extract the golden context under Tier-1 and recompute every stage-wise precision/recall and failure-mode bucket on the same 16,758 trajectories. The script is included in the artifact (`scripts/recompute_tier1_breakdown.py`); the recomputed per-trajectory data is released as `failure_mode_breakdown_tier1.csv` alongside the Tier-0 CSV.

Tier-1 expansion. On average, Tier-1 reference sets contain 11.7$\times$ more files than Tier-0 (1 patched file $\rightarrow \sim$12 files in the Tier-1 reference). The Tier-1 reference is therefore an *upper bound* on what an agent could reasonably need to inspect; Tier-0 is a

lower bound (only what was actually changed).

Effect on stage-wise recall and failure-mode buckets. Mean Search recall drops from 0.756 (Tier-0) to 0.292 (Tier-1) across all trajectories: agents rarely view every one-hop dependency, even on tasks they ultimately resolve. The cascading 0.5-threshold partition (Section 4.1) is sensitive to this drop: 4,379 trajectories that were assigned to downstream failure buckets under Tier-0 shift to the *Search* bucket under Tier-1, because their Search recall falls below 0.5 once measured against the larger denominator. The Tier-1 partition therefore concentrates almost all failure mass at Search and loses the discrimination among Read, Edit-Target, and Edit-Quality that Section 4.1 relies on.

What *does* survive under Tier-1. We verified two consistency properties: (i) every trajectory's Tier-1 recall is $\leq$ its Tier-0 recall at every stage, as required by set inclusion (0 violations across 16,758 trajectories); and (ii) the model ranking by aggregate recall is identical under Tier-0 and Tier-1 (Spearman $\rho = 1.00$). The decomposition is therefore robust to the choice of reference tier in the relative sense—which model is more recall-bound than which—but the absolute stage-wise rates and the failure-mode partition depend on the tier.

Why we report Tier-0 in the main paper. Two reasons. First, Tier-0 has an unambiguous semantics: a trajectory that fails to view a Tier-0 file failed to view a file the agent demonstrably needed to modify. Tier-1 includes files an agent *might* have benefited from viewing but did not strictly need; treating those as required inflates the apparent failure rate at Search and obscures the very downstream failure modes (Edit-Target, Edit-Quality) that motivate this paper. Second, Tier-0 is benchmark-portable: any benchmark with reference patches yields a Tier-0 reference for free, whereas Tier-1 requires a working call-graph extractor per language. Reporting Tier-1 as a sensitivity check rather than as the primary reference preserves portability and discriminative power simultaneously.

Why an intermediate "Tier-0.5" is also too coarse. A natural objection to the Tier-0 vs. Tier-1 dichotomy is that the right reference might lie between the two—e.g., the patched function plus all other functions in the same file. We tested this intermediate tier: the mean number of functions per gold file is 102, yielding a 66× expansion of the function-level reference set, larger than Tier-1's 11.7× file-level expansion. Under this intermediate tier, 94.5% of trajectories collapse into the Read-failure bucket, exceeding the Search-failure concentration produced by Tier-1 itself. The intermediate tier is therefore not a useful middle ground: any reference broader than "the functions actually modified" becomes too permissive at the function granularity TRAJEVAL operates on, because gold files typically contain many functions unrelated to the bug. This finding sharpens the design choice: function-level Tier-0 is the appropriate granularity for the search/read/edit decomposition, not because intermediate tiers were untested but because they fail the discrimination criterion empirically.

A.14 Failure-Mode Distribution: Remaining (Scaffold, Benchmark) Cells

This section extends the failure-mode breakdown introduced in Section 4.1 to the remaining five (scaffold, benchmark) cells. Every failed trajectory (Pass@1 = 0) is partitioned into one of four mutually exclusive, exhaustive buckets using a cascading 0.5 threshold on stage-wise recall: **Search** ($R_s < 0.5$), **Read** ($R_s \geq 0.5$, $R_r < 0.5$), **Edit-Target** ($R_r \geq 0.5$, $R_e < 0.5$), and **Edit-Quality** ($R_e \geq 0.5$). Rows sum to 100%.

Table 29: OpenHands on SWE-bench Verified.

Model	n	Search	Read	E.-Tgt.	E.-Qual.
GPT-5	158	11.4	3.2	17.7	**67.7**
GPT-5-mini	204	10.3	4.4	24.5	**60.8**
Qwen3-Coder-480B	196	9.7	10.2	15.3	**64.8**
Qwen3-Coder-30B	278	8.6	12.2	14.7	**64.4**
Qwen3-235B	276	14.9	9.8	10.9	**64.5**
Qwen3-32B	445	**41.8**	18.9	17.5	21.8
Qwen3-8B	444	**43.0**	19.8	12.6	24.5
Weighted avg.	2001	25.0	13.3	15.6	**46.0**

Table 30: LiveSWEAgent on SWE-bench Verified. Qwen3-8B and Qwen3-32B were excluded due to OpenRouter API errors with the bash-only LiveSWEAgent harness (Section 3, coverage note).

Model	n	Search	Read	E.-Tgt.	E.-Qual.
GPT-5	176	5.1	11.9	15.3	**67.6**
GPT-5-mini	214	8.4	13.1	15.4	**63.1**
Qwen3-Coder-480B	238	15.5	**40.3**	4.6	39.5
Qwen3-Coder-30B	386	33.4	30.3	3.1	33.2
Qwen3-235B	363	12.1	19.8	10.5	**57.6**
Weighted avg.	1377	17.2	24.3	8.8	**49.7**

Table 31: SWE-Agent on PolyBench Verified (also reproduced inline as Table 1, bottom).

Model	n	Search	Read	E.-Tgt.	E.-Qual.
GPT-5	208	18.8	**41.3**	8.7	31.2
GPT-5-mini	216	27.8	30.1	8.8	33.3
Qwen3-Coder-480B	261	28.0	**39.8**	11.1	21.1
Qwen3-Coder-30B	287	26.8	**36.6**	11.5	25.1
Qwen3-235B	295	**38.3**	25.8	11.2	24.7
Qwen3-32B	339	**61.4**	15.3	10.9	12.4
Qwen3-8B	353	**99.2**	0.6	0.3	0.0
Weighted avg.	1959	**47.0**	25.0	8.7	19.3

Table 32: OpenHands on PolyBench Verified.

Model	n	Search	Read	E.-Tgt.	E.-Qual.
GPT-5	202	19.8	**36.6**	9.9	33.7
GPT-5-mini	233	27.5	33.5	10.7	28.3
Qwen3-Coder-480B	244	24.2	**39.8**	5.7	30.3
Qwen3-Coder-30B	293	25.6	**38.9**	6.1	29.4
Qwen3-235B	301	30.9	33.6	7.6	27.9
Qwen3-32B	348	**52.9**	25.9	2.9	18.4
Qwen3-8B	361	**65.7**	18.6	4.4	11.4
Weighted avg.	1982	**37.9**	31.3	6.4	24.4

Table 33: LiveSWEAgent on PolyBench Verified. Qwen3-8B and Qwen3-32B were excluded due to OpenRouter API errors with the bash-only LiveSWEAgent harness (Section 3, coverage note).

Model	n	Search	Read	E.-Tgt.	E.-Qual.
GPT-5	239	16.7	**54.8**	4.6	23.8
GPT-5-mini	279	20.8	**53.8**	6.1	19.4
Qwen3-Coder-480B	297	31.0	**59.9**	1.3	7.7
Qwen3-Coder-30B	348	**50.6**	42.5	1.1	5.7
Qwen3-235B	329	24.0	**55.0**	3.0	17.9
Weighted avg.	1492	29.8	**52.8**	3.1	14.3

Cross-cell observations. Three patterns are visible across the six tables. (1) On SWE-bench, capable models (GPT-5 family, Qwen3-Coder-480B/30B) consistently exhibit Edit-Quality as the modal failure mode (60–70%), regardless of scaffold. (2) PolyBench shifts the modal failure category upstream: Search becomes the dominant mode for SWE-Agent (47.0% weighted avg.), and Read for LiveSWEAgent (52.8%), reflecting that multilingual repositories stress localization more than monolingual ones. (3) Smaller Qwen variants (8B, 32B) show consistently higher Search-failure rates than larger Qwen and GPT-5 models across every cell, supporting the interpretation that file localization is partially capability-bound.

A.15 Dataset Statistics

Table 34: Complete dataset statistics.

Metric	SWE-bench	PolyBench
Total instances	500	382
Languages	1 (Python)	4 (Py/Java/TS/JS)
Unique repositories	12	20
Avg files per patch	1.4	1.6
Avg functions per patch	2.1	2.4
Avg lines changed	42	38
Models evaluated	7	7
Agents evaluated	3	3
Total trajectories	9,500	7,258

A.16 Assumptions and Limitations

Reference Patch Assumption. We assume the reference patch represents the canonical solution. Our convergence analysis (93.7% exact function-level match, 100% partial overlap) covers 350 instances solved by multiple models. A key question is whether harder, unsolved instances exhibit greater solution diversity. We cannot directly measure this since unsolved instances lack successful agent patches to compare. However, SWE-bench instances were designed to have well-defined solutions verified by test suites, suggesting canonical fix locations exist even for hard instances. Future work could incorporate alternative reference patches where available in the dataset, enabling multi-reference evaluation that computes recall against the union of valid fix locations.

Function Extraction. Function extraction uses tree-sitter parsing with regex fallback for robustness. While tree-sitter achieves high accuracy on well-formed code, the regex fallback may miss

complex constructs such as nested functions, decorators, or metaprogramming patterns. The hybrid approach trades some accuracy for broad language support and handling of partial code snippets in trajectory observations.

Intervention Study Design. The two interventions in Section 4.3 have specific scope: (1) *Edit-commit checkpointing* is reported as a 5/5 existence proof on the EXACT-classified Near-Correct Corrupted cases (Appendix A.21); the realistic ceiling within Near-Correct Corrupted is bounded above by 30 (the EXACT + NEAR set) and remains an open empirical question, and a deployment-grade variant must gate the test invocation on cheaper signals to control cost. (2) *Parallel-sample consensus* is measured end-to-end on a single (model, scaffold, benchmark) cell—GPT-5 + SWE-Agent + SWE-bench Verified, n=500—and the $+3.0$ pp result is sub-threshold at $\alpha = 0.05$ ($p = 0.08$); replication on Qwen3-Coder-480B and on the OpenHands and LiveSWEAgent scaffolds, and a held-out instance set to rule out leakage from the leave-one-out construction, are immediate follow-ups.

Benchmark Scope. Our evaluation focuses on SWE-bench and PolyBench, emphasizing bug fixes in well-maintained open-source repositories. Generalization to other task types (security vulnerabilities, performance optimization) or codebases (proprietary, poorly documented) requires further validation.

A.17 Reproducibility and Artifact Release

We release the complete artifact at an anonymous repository link (Anonymous, 2026). The release is structured for direct reproduction of every figure and table in this paper:

- **Trajectory-extraction pipeline.** Per-architecture extractors for SWE-Agent (`view`/`str_replace`), OpenHands (`fsRead`/`fsWrite`), and LiveSWEAgent (bash-only). Tree-sitter parsers for Python, Java, JavaScript, and TypeScript with a regex fallback.

- **Per-trajectory feature CSV.** All 16,758 trajectories with stage-wise precision/recall, Pass@1, and assigned failure-mode bucket. A companion CSV (`failure_mode_breakdown_tier1.csv`) provides the same fields recomputed against the Tier-1 call-graph-expanded reference for the sensitivity analysis in Appendix A.13, along with the recomputation script (`scripts/recompute_tier1_breakdown.py`). This is the single source of truth for every quantitative claim in the paper.

- **Reproduction notebooks.** One per agent architecture, each running end-to-end from a small bundled trajectory sample to the corresponding row of Tables 6a and 1, so reviewers can verify the pipeline without re-executing 16,758 agent runs.

- **Checkpoint validation artifacts.** The 5 EXACT intermediate-edit patches submitted to the SWE-bench Docker harness (`intermediate_predictions_dedup.jsonl`), per-instance test reports, and the structured 5/5 pass result (`final_results.json`) for direct reproduction of Appendix A.21.

Random seed 42 is used throughout. The repository README documents which files reproduce which figures and tables in this paper.

A.18 Edit-Quality Taxonomy: Assignment Rule and Threshold Robustness

This appendix documents the cascading-precedence rule used to assign each Edit-Quality failure to exactly one sub-type (Section 4.2, Table 2), and reports a threshold-perturbation robustness check.

Assignment rule (MECE guarantee). Each Edit-Quality failure is assigned to exactly one sub-type via a cascading precedence rule applied to three computed signals: (1) *deletion overlap* (δ): fraction of gold-patch deletion lines found in the agent's intermediate edits on the gold file; (2) *token overlap* (τ): fraction of gold-patch addition tokens reproduced in the agent's intermediate edits; (3) *gold-file edit count* (k): number of distinct `str_replace` operations the agent performed on gold file(s) during the trajectory.

The assignment proceeds as follows, with each case consuming the trajectory so that exactly one label is assigned:

1. If $k = 0$ (agent never attempted an edit on the gold file): **Execution Gap**.

2. If $\delta > 0.5$ and $\tau > 0.5$: **Near-Correct Corrupted** (the agent produced content highly

similar to the gold patch, which was then either overwritten by a later edit or corrupted by a subsequent additive change).

3. If $\delta > 0.5$ and $\tau \in (0.2, 0.5]$: classify as **Wrong API Call** (if gold and agent invoke disjoint method sets), **Missing Edge Case** (if gold adds branch logic absent from agent's edit), or **Right Location, Wrong Logic** (default).

4. If $\delta \in (0.3, 0.5]$: **Incomplete Fix** (if gold is large and $k < 3$) or **Partial Understanding** (otherwise).

5. If $k \geq 3$: **Confused Thrashing** (multiple attempts, none converging).

6. Otherwise: **Wrong Branch Point** (if both gold and agent add control flow), **Wrong API Call** (if disjoint method sets), or **Wrong Function in File** (default).

This cascading structure ensures mutual exclusivity: higher-overlap cases are consumed first (steps 2–4), then multi-attempt cases (step 5), with the remainder assigned by structural features (step 6). Thresholds ($\delta = 0.5$, $\tau = 0.5$, $k = 3$) were selected via single-annotator inspection of a 50-trajectory calibration set. Inter-annotator reliability of the cascading rule's labels is reported separately in Appendix A.19.

Threshold robustness. To verify that the qualitative findings of Section 4.2 are not artifacts of the chosen thresholds, we recomputed the theme distribution under ± 0.1 perturbation of δ and τ and ± 1 perturbation of k (eight off-center configurations in addition to the default). Across all configurations:

- Coherence Collapse remains the largest or second-largest theme, ranging 36–56%.
- Scope/Completeness remains $< 10\%$ in every configuration.
- The relative ordering of model-level Coherence Collapse rates (Qwen3-32B largest, GPT-5/GPT-5-mini smallest) is preserved.

The Near-Correct Corrupted sub-type's exact-match validation (Appendix A.20) is independent of these thresholds—it operates on diff-hunk reconstruction against the gold patch's structure—and is therefore unaffected.

A.19 Inter-Annotator Agreement on the Cascading Rule

This appendix reports inter-annotator reliability of the Edit-Quality cascading rule's category labels (Section 4.2, Appendix A.18).

Procedure. A second author independently labeled 100 SWE-bench Verified Edit-Quality failures, sampled uniformly at random from the 914-failure population. The annotator received only the agent trajectory (action history with observations) and the gold patch; they did not see the cascading rule's prior labels, the computed deletion/token/edit-count signals, or any other automated output. The annotator was given the category definitions from Appendix A.18 (steps 1–6) and assigned each trajectory to exactly one category.

Pre-registration. The cascading rule structure (steps 1–6 of Appendix A.18) and the threshold values ($\delta = 0.5$, $\tau = 0.5$, $k = 3$) were finalized before the second annotator began labeling; thresholds were not adjusted in response to the IAA outcome. The threshold-perturbation robustness check in Appendix A.18 was likewise computed independently of the IAA labels. This separates the rule design from its reliability assessment: the second annotator's labels test the rule, they did not shape it.

Multi-level reporting. We report agreement at three levels of granularity, chosen to match the paper's claim hierarchy rather than to maximize κ.

- **3-class (headline).** Coherence Collapse vs. Semantic-or-Mislocalization vs. Execution Gap. This is the granularity at which the abstract and conclusion make claims (e.g., "Coherence Collapse is the largest single theme"). $\kappa = 0.80$**, 90% raw agreement.**
- **5-class (refinement).** 3-class scheme, plus separating Scope/Completeness from Semantic Error and preserving the two Coherence Collapse sub-types (Confused Thrashing vs. Near-Correct Corrupted). $\kappa = 0.71$**, 83% raw agreement.**
- **6-category (full cascading rule).** The complete category set used in Table 2. $\kappa = 0.68$**, 77% raw agreement.**

By the Landis–Koch convention, $\kappa = 0.80$ is the upper end of substantial (with "almost perfect"

starting at 0.81); $\kappa = 0.71$ is substantial; $\kappa = 0.68$ is at the lower end of substantial. The 3-class number is the one the paper's headline claim depends on.

Disagreement analysis. Of the 23 fine-grained disagreements, 15 (65%) involve the Scope/Completeness category—the rule's most contentious decision boundary, where step 4 of the cascading rule itself acts as a tie-break between Incomplete Fix and Partial Understanding. With direction tracked, the 15 Scope-related disagreements decompose as: 6 Rule=Scope/Human=Near-Correct Corrupted, 4 Rule=Scope/Human=Semantic Error, 3 Rule=Scope/Human=Confused Thrashing, and 2 Rule=Semantic/Human=Scope. The 6 Scope-to-NCC and 3 Scope-to-Thrashing disagreements are notable: in 9 of the 15 Scope-involving cases, the human annotator labels a trajectory as Coherence Collapse where the cascading rule labels it as Scope/Completeness, never the reverse. The 5 EXACT Near-Correct Corrupted cases on which the paper's checkpoint validation depends (Appendix A.20) are validated by diff-hunk reconstruction and direct test-suite execution, both independent of the cascading rule's labels.

At the 3-class level, the 10 remaining disagreements all fall in the same direction: the cascading rule labels the trajectory in the Semantic-or-Mislocalization-or-Scope zone, while the second annotator reads it as Coherence Collapse. This directional asymmetry has a substantive interpretation: the cascading rule *understates* rather than overstates Coherence Collapse. Reassigning all 10 disagreements to match the human annotator would only *increase* the Coherence Collapse rate; the headline "Coherence Collapse is the largest single theme" is robust to either labeling choice, and arguably more so under the human's stricter classification.

Why we report all three levels. A single κ number masks where reliability is high vs. low. The full 9-sub-type taxonomy is retained in the paper's analysis for descriptive granularity, but the IAA we can defend most strongly is at the granularity of the central claim (3-class). Reporting all three levels allows readers to judge each claim against the appropriate reliability threshold.

A.20 Near-Correct Corrupted: Diff-Hunk Validation

The 97 trajectories assigned to Near-Correct Corrupted by the rule above (Section 4.2) all exceed token-overlap thresholds $\delta > 0.5$ and $\tau > 0.5$. To strengthen this beyond bag-of-tokens overlap, we additionally ran a diff-hunk reconstruction check: for each case, we extracted the agent's intermediate edit at the moment of highest token overlap with the gold patch, normalized whitespace and stripped imports/comments, and compared its diff hunks against the gold patch's diff hunks. The result is reported in main-body Table 3: 30/97 (31%) reconstruct gold-patch hunks at the strong-match level (EXACT or NEAR); 96/97 (99%) target the gold patch's code region (everything except the single DIFFERENT case); the median gap between the near-correct edit and its overwriting/corrupting edit is 6 trajectory steps; and of the 5 EXACT cases, GPT-5 accounts for the largest share of strong-match cases per Edit-Quality failure (57%), consistent with the broader pattern that capable models reach correct fixes more often before losing them.

A.21 Edit-Commit Checkpoint Validation

This appendix reports the validation underlying Prediction 1 (Section 4.3). The 5 EXACT-classified intermediate edits identified by the diff-hunk validation (Appendix A.20) are by construction bit-identical (modulo whitespace) to the gold patch's hunks. We tested whether they actually pass the existing test suite, which would establish edit-commit checkpointing as a deployable intervention rather than a hypothetical one.

Setup. For each of the 5 EXACT cases, we extracted the agent's intermediate edit at the trajectory turn of highest gold-token overlap, reconstructed the resulting candidate patch, and submitted it to the official SWE-bench Verified Docker harness. The harness applies the patch to the pre-bug commit and runs the full instance test suite; a case is considered recovered if and only if all FAIL_TO_PASS tests pass and no PASS_TO_PASS tests regress.

Result. All 5 EXACT cases pass the full SWE-bench test suite. This confirms that these agents produced a provably correct solution mid-trajectory that was subsequently corrupted or overwritten by a later edit. An edit-commit checkpoint that freezes

test-passing intermediate edits would have recovered all 5 instances. We report this as a strict existence proof—the 5 EXACT cases are the subset where the intermediate edit is bit-identical to the gold patch and demonstrably passes the full test suite, establishing edit-commit checkpointing as a deployable intervention rather than a hypothetical one. The realistic ceiling within Near-Correct Corrupted is bounded above by 30 (the EXACT + NEAR set, 31% of the 97 NCC cases) and remains an open empirical question.

Cost model. Edit-commit checkpointing is not free. A naive implementation runs the FAIL_TO_PASS test subset after every `str_replace` on a gold-area file, which can multiply per-trajectory test invocations by 10–40× depending on edit count and test-suite duration. Practical deployments would gate the test invocation on cheaper signals (e.g., diff-similarity to prior edits, or a learned predictor over edit content) to avoid running the harness on edits unlikely to be improvements. We report the 5/5 result under the strictest possible cost assumption (test every candidate edit) so that the existence-proof claim is not contingent on cost-saving heuristics; deployment-grade variants are an open engineering question.

Scope and limitations. The validation establishes a strict lower bound: the 5 EXACT cases are the subset where the intermediate edit is bit-identical to the gold patch and passes tests *by construction*. We attempted broader validation on the 25 NEAR cases but our current diff-reconstruction pipeline introduces patch-application errors (missing line-number context) that confound semantic-correctness measurement; we leave end-to-end agent re-execution under checkpointing to follow-up work, where the agent would be re-run with a real test-success-triggered freeze rather than retroactively evaluated. The 5 EXACT cases are sufficient to establish that edit-commit checkpointing is a deployable, not merely hypothetical, intervention; the realistic ceiling within Near-Correct Corrupted is bounded above by 30 (the EXACT + NEAR set) and remains an open empirical question.

A.22 Consensus-as-Relevance-Signal Validation

This appendix reports the validation underlying Prediction 2 (Section 4.3). The oracle-guided intervention in Section 4 requires the reference patch at execution time and is therefore not deployable. We test whether a non-oracle relevance signal, computed from parallel-sample agreement among models alone, recovers a comparable lift.

Signal definition. For each SWE-bench Verified instance i and threshold $N \in \{2, 3, 4, 5\}$, we define

$$\textsc{Consensus}_N(i) = \{f \mid f \text{ viewed by} \geq N \text{ of the 6 non-GPT-5 m}$$

i.e., the leave-one-out set of files that at least N of the other six models inspected during their trajectories on instance i. The signal uses no reference patch and no information from GPT-5 itself.

Recovery test. Of the 500 SWE-bench Verified instances, the oracle-guided intervention (Section 4) would fire on 490 (the cases where GPT-5 viewed at least one gold file). For each of these 490 instances we ask: does $\textsc{Consensus}_N$ contain at least one gold file? If yes, the consensus signal would have triggered the same confirmatory feedback that produced the +4.6 pp oracle gain.

Results. Table 35 reports precision and recall of $\textsc{Consensus}_N$ against gold files (averaged across 500 instances), the fraction of the 490 oracle-firing cases that consensus also covers, and the implied lower bound on Pass@1 lift (recovery × 4.6 pp).

Table 35: Consensus signal validation. For each threshold N, we compute the set of files viewed by $\geq N$ of the 6 non-GPT-5 models on each instance. Precision/recall are against gold edit locations. "Oracle recovered" is the fraction of the 490 instances where the oracle would fire that consensus also covers at least one gold file. Analytical ceiling = recovery rate × 4.6 pp (the oracle gain), achieved only under perfect-precision triggering.

N	Precision	Recall	Oracle recovered	Analytical ceiling
2	35.2%	95.5%	99.0% (485/490)	+4.6 pp
3	52.4%	94.2%	97.3% (477/490)	+4.5 pp
4	66.8%	91.5%	92.4% (453/490)	+4.3 pp
5	75.6%	86.2%	82.4% (398/483)	+3.8 pp

Analytical interpretation. At $N{=}3$, $\textsc{Consensus}_N$ recovers 97.3% of oracle-triggering cases. Multiplying by the $+4.6$ pp oracle ceiling implies an analytical ceiling of $+4.5$ pp Pass@1 lift, achieved only if every consensus-driven trigger produced the same gain as an oracle-driven trigger (which requires perfect-precision triggering—a condition the 52.4% precision at $N{=}3$ does not meet). Increasing N trades recovery for precision:

at N=5, precision rises to 75.6% but recovery drops to 82.4%, suggesting N=3 is a reasonable operating point.

End-to-end measurement. The end-to-end consensus measurement is reported in main-body Table 5 (Section 4.3). At CONSENSUS$_3$ precision of 52.4%, the gain/loss decomposition (39 gained, 24 lost, net +15) reflects that roughly half of the confirmatory triggers occur on files outside the gold set; the agent occasionally over-weights these confirmations and edits the wrong location. This identifies a concrete mechanism for the gap to the analytical bound and suggests a direct improvement path: combining CONSENSUS$_N$ with a higher-precision filter (e.g., N=4 or a learned re-ranker over consensus-flagged files) to suppress losses without sacrificing gains.

Implications. The result establishes that the Edit-Quality bottleneck identified by TRAJEVAL is not gated on having a reference patch at deployment. The function-level convergence we observe (93.7% on multiply-resolved instances, Section 4) is empirically strong enough that consensus among independent samples reproduces the oracle's confirmatory signal almost entirely. Self-consistency decoding (Wang et al., 2023) is the chain-of-thought analogue; here it operates on tool-use trajectories rather than token sequences. Two caveats apply: (i) consensus requires running $\geq N{+}1$ samples, which has compute cost (this matches self-consistency); (ii) the validation is conducted on the same 500 SWE-bench Verified instances on which the per-model trajectories were collected, so we cannot rule out leakage from instance-level features into the consensus signal—though the leave-one-out construction (excluding GPT-5 itself) and the fact that no reference patch enters the signal partially mitigate this. Re-running consensus on a held-out instance set, and tightening the gap between the empirical $+3.0$ pp measurement and the analytical $+4.5$ pp ceiling via higher-precision triggering, are immediate follow-ups.

A.23 Trajectory Length and Coherence Collapse

A natural concern is whether Coherence Collapse is simply long-context degradation under a different name—agents that run longer exhaust their context window and produce incoherent output. We test this by computing Spearman rank correlations between trajectory length (number of history steps) and failure sub-type assignment across all 914 Edit-Quality failures (SWE-Agent, SWE-bench Verified).

Results (Figure 2). Coherence Collapse shows a moderate positive correlation with trajectory length ($\rho = 0.32$, $p = 3.4 \times 10^{-23}$): the Q4 (longest) quartile has a 63.7% collapse rate versus 21.7% in Q1 (shortest), a 42 pp difference. However, the sub-type decomposition reveals a critical dissociation: *Near-Correct Corrupted* shows **no correlation** with trajectory length ($\rho = 0.001$, $p = 0.98$), maintaining a stable ∼10–11% rate across all quartiles. The correlation is driven almost entirely by *Confused Thrashing* (the multi-attempt variant; $\rho = 0.34$, $p = 8.5 \times 10^{-27}$), which rises from 10.5% in Q1 to 52.9% in Q4.

Among the 97 Near-Correct Corrupted cases, the gap between the near-correct intermediate edit and its corruption is strongly correlated with total trajectory length ($\rho = 0.77$, $p < 10^{-14}$, $n = 64$), indicating that longer trajectories introduce more intervening steps before corruption—but do not increase the *probability* of corruption.

Interpretation. Coherence Collapse is partially but not entirely explained by long-context effects. The Confused Thrashing variant (29.1% of Edit-Quality failures) is length-dependent—these agents likely run longer *because* they are thrashing, not the reverse (the causal direction is confounded by the data). The Near-Correct Corrupted variant (10.6%) is length-independent, suggesting a qualitatively different failure mechanism: the agent produces a correct fix early in the trajectory regardless of total length, then corrupts it through overconfident refinement. This dissociation indicates that context-window degradation is not the sole driver of Coherence Collapse and supports the taxonomy's distinction between the two sub-types.

A.24 Edit-Quality Worked Examples

We present three trajectories illustrating the dominant Coherence Collapse sub-types, drawn from our analysis of the 914 Edit-Quality failures.

A.24.1 Worked Example 1: Confused Thrashing

Instance: `django__django-11848` **Model:** GPT-5 **Gold file:** `django/utils/http.py`

Gold patch (two-year window fix for HTTP date parsing):

```
-           if year < 70:
```

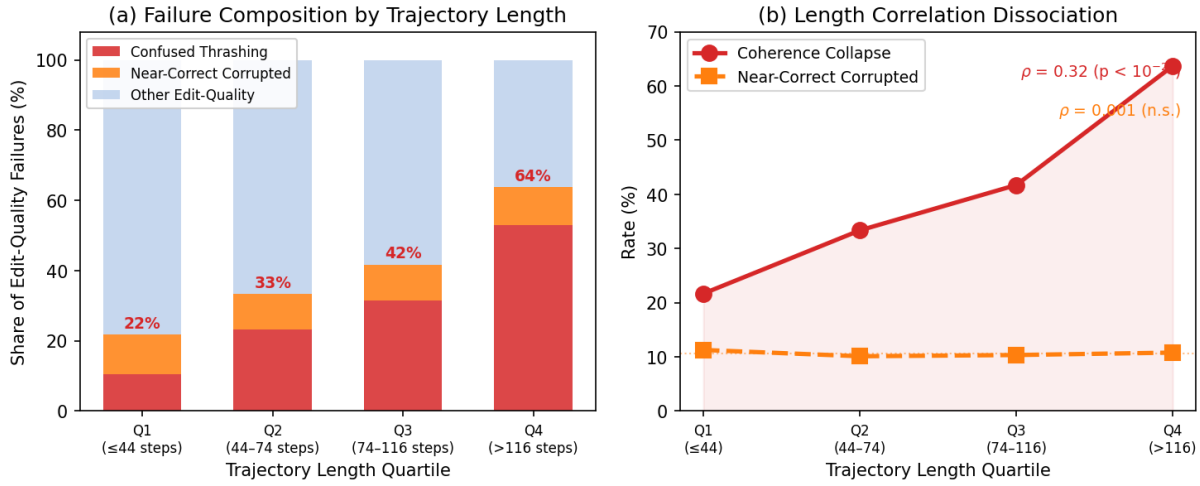

Figure 2: Relationship between trajectory length and Coherence Collapse. (a) Failure theme composition by trajectory-length quartile; Coherence Collapse (red+orange) grows from 22% to 64% (aggregate $\rho = 0.32$). (b) Dissociation: Confused Thrashing (red) scales with length ($\rho = 0.34$) while Near-Correct Corrupted (orange) is flat ($\rho = 0.001$).

```
-                 year += 2000
+             current_year = datetime.datetime.utcnow().year
+             current_century = current_year - (current_year % 100)
+             if year - (current_year % 100) > 50:
+                 year += current_century - 100
              else:
-                 year += 1900
+                 year += current_century
```

Agent trajectory (4 edits on gold file, none preserved):

```
Turn 22: str_replace http.py
  OLD: "if year < 70: year += 2000 else: year += 1900"
  NEW: (replaces with month/day extraction logic -- wrong approach)

Turn 28: str_replace http.py
  OLD: (reverts and refactors year parsing)
  NEW: "year_str = m.group('year'); year = int(year_str)..."
       (adds variable but loses the century-window logic)

Turn 34: str_replace http.py  [FAILED -- old_str not found]

Turn 42: str_replace http.py  [FAILED -- old_str not found]
```

Final patch: Edits `django/db/migrations/serializer.py` (completely unrelated file).

Diagnosis: The agent found the correct file and twice attempted the correct code region. However, each edit overwrote the previous without converging on the gold logic. After two failed `str_replace` attempts (stale `old_str` due to prior edits), the agent abandoned the approach entirely and pivoted to an unrelated fix. This is the signature *Confused Thrashing* pattern: the correct location is reached but coherent multi-step reasoning collapses.

A.24.2 Worked Example 2: Near-Correct Corrupted (Additive)

Instance: `matplotlib__matplotlib-20859` **Model:** GPT-5 **Gold file:** `lib/matplotlib/legend.py`

Gold patch (support legends on SubFigure by importing FigureBase):

```
-         from matplotlib.figure import Figure
+         from matplotlib.figure import FigureBase
          ...
-         elif isinstance(parent, Figure):
+         elif isinstance(parent, FigureBase):
```

Agent trajectory:

```
Turn 48: str_replace legend.py            <-- CORRECT
  OLD: "from matplotlib.figure import Figure"
  NEW: "from matplotlib.figure import FigureBase"

Turn 70: str_replace legend.py
<-- CORRUPTS (additive)
  OLD: "elif isinstance(parent, Figure): ..."
  NEW: "elif isinstance(parent, FigureBase):
            self.set_figure(parent.figure)  ..."
       (adds .figure indirection -- not in gold)
```

Final patch: Contains *both* edits, but the turn-70 change adds `parent.figure` instead of the gold's direct reference, causing the test suite to fail.

Diagnosis: At turn 48, the agent produces an edit that *exactly matches* the gold patch's first hunk. At turn 70, a second edit is semantically close but adds an extra `.figure` dereference not present in the gold. The import change alone would have sufficed; the over-correction at turn 70 introduced the failure. This is the *additive corruption* variant of Near-Correct Corrupted: the original correct edit is preserved, but a subsequent well-intentioned refinement breaks the patch.

A.24.3 Worked Example 3: Near-Correct Corrupted (Destructive Overwrite)

Instance: `django__django-15277` **Model:** Qwen3-32B **Gold file:** `django/db/models/fields/__init__.py`

Gold patch (guard `MaxLengthValidator` against `None`):

```
-        self.validators.append(validators.MaxLengthValidator(self.max_length))
+        if self.max_length is not None:
+            self.validators.append(validators.MaxLengthValidator(self.max_length))
```

Agent trajectory:

```
Turn 34: str_replace fields/__init__.py
<-- EXACT GOLD EDIT
  OLD: "self.validators.append(validators.MaxLengthValidator(self.max_length))"
  NEW: "if self.max_length is not None:
            self.validators.append(validators.MaxLengthValidator(self.max_length))"

Turn 48: str_replace fields/__init__.py
<-- DESTRUCTIVE OVERWRITE
  OLD: "self.db_collation = db_collation
        self.validators.append(validators.MaxLengthValidator(self.max_length))"
  NEW: "if self.max_length is not None:
            self.validators.append(validators.MaxLengthValidator(self.max_length))"
       (deletes 'self.db_collation = db_collation' -- breaks CharField)
```

Final patch: Only contains a `reproduce_error.py` script; the source-file edits cancelled out or broke the build, yielding no productive diff.

Diagnosis: At turn 34, the agent produces a patch *identical* to the gold reference (`del_recall`=1.00, `add_recall`=1.00, `add_seq`=1.00). We confirmed via the SWE-bench Docker harness that the turn-34 patch passes the full test suite (Appendix A.21). Fourteen steps later, the agent re-attempts the same logical fix but with a larger `old_str` context that inadvertently deletes the `db_collation` assignment. This destroys the correct turn-34 edit and breaks `CharField` initialization. The agent had already solved the problem—the failure was caused purely by an unnecessary follow-up edit. This is the *destructive overwrite* variant of Near-Correct Corrupted: the test-passing intermediate edit is deleted and replaced by a corrupted version.