EDIT-INDUCED DEPENDENCY MISMATCH: PROPAGATION OMISSIONS, RECOVERY, AND EVALUATION BLIND SPOTS IN CODING AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Dependency mismatches induced by code edits can cause task evaluation failures or leave unintended defects along existing consumer paths even when the evaluation passes. Focusing on consumer obligations introduced by edits and existing consumers left unsynchronized, we manually audit 2,879 archived `deepseek-v4-flash` runs across 64 DeepSWE tasks. Three human reviewers each independently reviewed the entire corpus under blinded conditions, establishing a human-confirmed terminal diagnosis set of 473 runs (16.4%), explicitly including disputed-scope cases, across 37 tasks and 85 distinct within-task consumer relations. Of these runs, 177 have execution-failure evidence and 296 have static reachability evidence, with 0 unresolved evidence-source cases; 290 have undisputed scope for all terminal events, and 158 of these have execution support for at least one terminal event. Recovery events also occur in 370 runs. Five local DeepSWE replays show that synchronizing the diagnosed consumers restores the target behavior while the listed controls continue to pass. In the Kysely case, the original task evaluation passes 276/276 tests, yet a new field is still lost during plugin transformation; adding a single line to copy it restores the semantics. From 500 archived SWE-bench Verified runs, three targeted passing patches expose propagation gaps involving glossary keys, range constraints, and coordinate attributes through execution. Three terminal diagnoses and four recovery runs on FeatureBench extend the evidence to Qwen3.8-Flash. These cases support measuring task scores, terminal propagation states, and recovery separately.

1 INTRODUCTION

Repository-level coding agents complete software engineering tasks through repeated reading, editing, and testing (Yang et al., 2024; Wang et al., 2025). An edit not only introduces new code but can also impose new obligations on unchanged code: when a constructor gains a field, an existing transformer must preserve it; when a query return value gains a wrapper, other execution paths must recognize it. Such mismatches can cause the current task evaluation to fail, or escape evaluation and leave unintended defects in patches that receive passing scores.

Query's infinite-query path fails to consume a new return wrapper: synchronizing the shared consumer entry point raises the original evaluation from 49/50 to 50/50, preserving ordinary-query restoration and new-request controls. Kysely passes all 276/276 original tests, yet its new NULLS modifier is lost in an existing transformer. Direct compilation preserves the modifier while plugin transformation loses it; adding `nulls: node.nulls` restores the behavior and preserves the original evaluation (Figure 1). These cases expose consumer gaps both detected and missed by the original evaluations.

We call this phenomenon **edit-induced dependency mismatch**. Our focus is on consumer obligations introduced by an agent's own edits and existing consumers that fail to meet them. This phenomenon has two characteristics. First, the mismatch concerns a program relationship: unchanged consumer source code does not guarantee that its behavior remains appropriate for new inputs. Second, the mismatch has a state over the course of execution: a gap may be repaired later, remain in the final patch, or surface only under additional checks of composed behavior. Studying

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

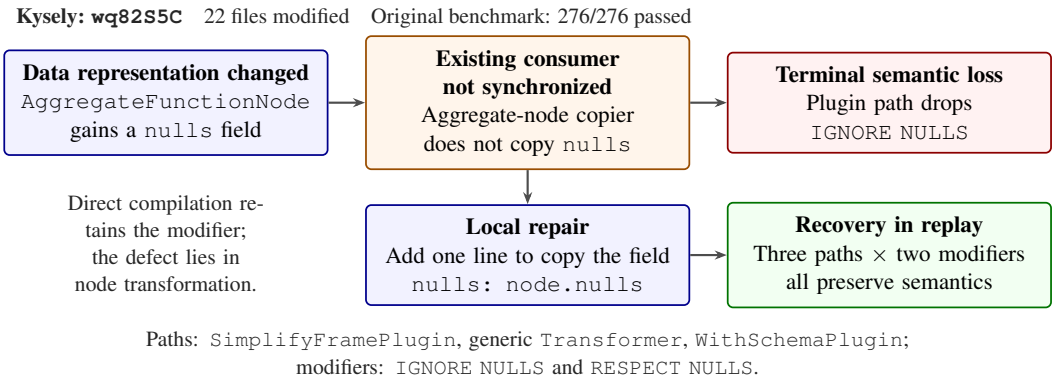


Figure 1: Kysely trial `wq82S5C`: a new field takes effect on the direct path but is discarded by an existing transformer. The original evaluation score comes from the run archive; the before- and after-repair results come from an independent AST/SQL-generation replay. Six transformation checks in the same run identify the field omission.

only failing runs misses gaps among passing runs; examining only final patches obscures events that have already been repaired.

Change-impact analysis and dependency planning have long studied how to propagate code edits (Bohner & Arnold, 1996; Bairi et al., 2024). We connect edits in naturally occurring runs, specific consumers, intermediate recovery, and final evaluations into verifiable evidence. Similar patches may arise from localization failures, code misunderstanding, preservation of old behavior, or omissions of historical information; program inconsistencies can be tested through execution, whereas cognitive attribution requires different evidence.

We organize the study around three questions: **RQ1**: How can propagation omissions be identified and distinguished from ordinary implementation errors? **RQ2**: How do terminal omissions, recovery during the trajectory, and original evaluation outcomes relate to one another? **RQ3**: With the model-generated patch held fixed, can synchronizing the diagnosed consumers restore the corresponding behavior? We make four contributions: an operational definition based on an "edit–obligation–consumer–diagnosis" chain; a complete audit of 2,879 runs with recovery/evaluation cross-tabulations, supplemented by terminal and recovery cases from another model on FeatureBench; five DeepSWE replays and three targeted SWE-bench Verified counterexamples that test the behavioral effects of local synchronization with patches and inputs held fixed; and, through these targeted replays, confirmed instances in both DeepSWE and SWE-bench Verified where the original evaluation passes despite consumer defects, revealing blind spots in the original task evaluations for these instances.

2 Study Focus and Diagnostic Boundaries

2.1 From Source Edits to Consumer Obligations

For each event, we record four pieces of evidence, $e = (\Delta, O, C, D)$: Δ is a new representation, interface, state, or configuration already written into the source; O is an adaptation obligation introduced by the edit and supported by task requirements or existing program semantics; C is a specific consumer that handles the relevant input; and D is a diagnosis showing that this consumer fails to meet the obligation. For example, in Kysely, Δ is the new `nulls` field, O is the requirement for node transformations to preserve user-specified modifiers, C is an existing aggregate-node transformation branch, and D is the difference in attributes and SQL before and after transformation. A field's optional status in the type system does not eliminate its semantic obligation in a given invocation.

Table 4 provides a unified decision table: the edit must have been applied, and the consumer must predate the patch and be distinct from newly written logic and the modified function's own contract; the obligation must be supported by an explicit task statement or an existing contract that still

applies, and reachability must specify an entry point, input/state, branch conditions, and consumer chain. We exclude a relationship if it clearly fails to meet these conditions; if the obligation or path cannot be established, we retain it as unresolved rather than judging it incorrect merely because it differs from the reference patch. When the obligation and diagnosis are established but the triggering scenario's inclusion in task scope remains disputed, we retain the diagnosis and record a separate scope flag. Diagnostic evidence is classified as execution output or concrete static reasoning. An existing path in the same file can also qualify when it has a distinct processing role. Detailed decision rules and boundary cases appear in Appendix A.2.

2.2 Terminal States, Recovery, and Program-Level Confirmation

A **terminal propagation omission** means that the corresponding gap remains in the final patch; a **recovery event** means that a distinct behavioral failure of an existing consumer occurs in the trajectory and is subsequently repaired. A single run may recover from one event while leaving another unresolved. Wink monitors coding-agent trajectories and supplies asynchronous guidance to recover from misbehaviors (Nanda et al., 2026). Temporary inconsistencies between planned consecutive edits, migration-time compilation diagnostics, and synchronization of generated files or test snapshots alone fall outside this definition. Alongside $e = (\Delta, O, C, D)$, we record each identified event's terminal or recovered state. Terminal events also carry evidence $E \in \{\text{exec}, \text{static}\}$ and scope $S \in \{\text{undisputed}, \text{disputed}\}$ labels (Appendix B.2).

Program-level replay confirmation further fixes the original final patch and check inputs and compares behavior before and after local synchronization. The local program diagnosis is supported only if the target behavior is restored and the listed controls continue to pass. Repairs are restricted to the diagnosed consumer set and supporting locations needed to maintain its shared invariants, while preserving the new functionality, execution paths, check inputs, and assertions. The repair set may span multiple consumer locations; we assess its local behavioral effects through target recovery on the listed inputs, control checks, and changes in non-target behavior.

2.3 Relationship to Prior Work

Surveys and empirical trajectory studies characterize software-engineering agents and their success and failure behaviors (Liu et al., 2025; Majgaonkar et al., 2025). SWE-bench provides execution-based tasks, while SWE-agent and OpenHands provide agent frameworks (Jimenez et al., 2024; Yang et al., 2024; Wang et al., 2025); RepoCoder, RepoGraph, and Agentless study retrieval, code structure, and repair workflows, respectively (Zhang et al., 2023; Ouyang et al., 2025; Xia et al., 2025). DeepSWE examines incomplete patches accepted by original tests (Huang et al., 2026b); SWE Refactor Bench separately evaluates migration completeness and behavioral correctness (Hong et al., 2026).

CodePlan, AutoCodeRover, and CodeR use dependency analysis, program structure, or task graphs to organize edits (Bairi et al., 2024; Zhang et al., 2024; Chen et al., 2024). Aider provides repository maps, and RefactorBench studies stateful reasoning in multi-file refactoring (Aider contributors, 2026; Gautam et al., 2025). SWE-Touch tests adaptation to simulated external edits and conflicting messages (Tan et al., 2026). MAGIS uses multi-role quality review, while SWE-Search uses tree-search feedback (Tao et al., 2024; Antoniades et al., 2025). Our event-level audit links edit-induced consumer obligations, terminal omissions, natural recovery, and original scores, with local synchronization testing specific behavioral effects.

3 Data and Audit Method

3.1 Fixed Run Corpus

The tasks come from the DeepSWE benchmark and its public repository (Huang et al., 2026b;a). We define the complete corpus using the unique run identifiers in the archived score table: 64 tasks, 55 repository snapshots, and $2{,}879$ scored runs. The original design called for 15 repetitions per task per condition, totaling $2{,}880$ runs; one run did not enter the official scored set. Additional archived directories do not increase the denominator. Every included record can be linked to a final patch and

Table 1: Corpus and run settings. DeepSWE provides the tasks; coding-run settings and the manual audit procedure are reported separately.

Item	Archived facts and measurement units
Complete corpus	2,879 runs, 64 tasks, 55 repository snapshots
Original evaluation outcomes	1,286 passing; 1,593 failing; original values retained
Final patches	Median of 9 modified paths; 2,868 runs modify at least two files; 11 patches are empty
Coding-run stack	`deepseek-v4-flash`; `mini-swe-agent 2.4.6`; Harbor `0.21`
Original conditions	`bare`, `inject-file`, `guided`; 15 runs planned per task per condition
Audit procedure	Three independent full-corpus blinded human reviews plus a separate independently sampled blinded audit; final labels determined by humans
Statistical units	Runs, task coverage, diagnosed events, and within-task consumer relation types, reported separately

a complete normalized trajectory, with 1,286 passing and 1,593 failing original evaluation outcomes (Table 1).

The full model identifier is `openai/deepseek-v4-flash`, whose prefix is a provider tag used by the execution framework. The three original conditions provide no repository index, provide an index, and add a usage protocol to the index, respectively; these are run-level settings. Modified-path counts measure patch breadth across source, tests, documentation, and configuration; consumer relations are counted separately. Parameters follow the default benchmark configuration.

3.2 Full-Corpus Blinded Review and an Independently Sampled Blinded Audit

We used annotation guidelines preregistered before exploratory analysis. Three reviewers with experience in large language models and code analysis each independently reviewed all 2,879 runs under blinded conditions, establishing edit–obligation–consumer–diagnosis evidence chains from task requirements, final patches, execution records, and trajectories. The full-corpus review hid original score summaries, existing audit labels, and the other reviewers' rationales while retaining execution evidence needed for diagnosis. Before discussion in this stage, all-three-reviewer agreement was 97% for the primary category and 89% for complete event tuples $e = (\Delta, O, C, D)$, covering both terminal and recovery events. Disagreements were resolved through evidence re-examination and discussion. Primary labels, terminal states, and events were recorded separately; "unresolved" is the reviewers' agreed classification for insufficient diagnostic evidence.

In addition to the full-corpus blinded review, we conducted an independently sampled blinded audit with the original task-evaluation score summaries and existing audit labels hidden. Reviewers judged cases independently using task requirements, source code, original model patches, and relevant execution evidence; initial annotations were fixed before unblinded comparison and adjudication of disagreements.

Aggregate statistics are recomputed from the final human-audit records covering all 2,879 runs; five local replays provide execution evidence for the corresponding relations. Appendix A describes the audit chronology and final criteria; Appendix E identifies the submitted data and diagnosis records.

3.3 Program State and Model-Visible Information

Program-level attribution and process interpretation require different evidence. Source code, patches, and raw execution outputs can support judgments about program behavior; claiming that the model "read a consumer," however, requires locating the relevant code in the tool messages actually presented at the time. Complete archived outputs may retain sections omitted from those messages and are used for program analysis. The focal cases therefore separately record three things: whether relevant information was presented, whether an explicit judgment was made, and whether synchronization and validation were performed. Explicit judgments provide process evidence at the message level. Audit sources and final criteria appear in Appendix A; the submitted-materials index is in Appendix E.

Table 2: Terminal diagnoses (including disputed scope), evidence, and recovery. Any exec.: at least one execution-supported terminal event; Static only: all terminal events statically supported. Scope filters nest $473 \rightarrow 290 \rightarrow 158$ runs: all diagnosed, all terminal events undisputed, then at least one execution-supported event. Original evaluation outcomes are retained.

(a) Terminal diagnoses by scope, evidence, and original outcome

Original outcome	All runs	Diagnosed	Any exec.	Static only	All events undisputed	Undisputed + execution
Passed	1,286	146	11	135	75	5
Failed	1,593	327	166	161	215	153
Total	2,879	473	177	296	290	158

Terminal diagnoses: 37 tasks / 85 consumer relation types / 473 runs / 546 events

(b) Terminal status and recovery events

Terminal status	With recovery events	No recovery event recorded	Total
Terminal diagnosis	83	390	473
Other (including unresolved)	287	2,119	2,406
Total	370	2,509	2,879

4 Relationships among Propagation Omissions, Recovery, and Evaluation Outcomes

4.1 Terminal Labels Cover Both Passing and Failing Runs

For run r, let T_r contain its diagnosed terminal events and $A_r = \mathbf{1}[T_r \neq \varnothing]$. With $R = 2{,}879$, the run share is $p_{\text{run}} = R^{-1} \sum_{r=1}^{R} A_r = 473/2{,}879 = 16.4\%$, including disputed-scope diagnoses. Task coverage is $37/64 = 57.8\%$. Among 546 diagnosed terminal events, execution and static evidence account for $179/546 = 32.8\%$ and $367/546 = 67.2\%$. These events form 85 relation types after within-task deduplication. The nested scope subsets contain 290 runs with all terminal events undisputed (10.1% of R), and 158 of those with at least one execution-supported terminal event (5.5%), including 5 originally passing and 153 failing runs. Across all diagnoses, 177 runs contain at least one execution-supported terminal event and 296 have only statically supported terminal events; 29 runs mix the two event types, and each event retains its own label. Evidence-unresolved cases number 0. Appendix B.2 gives the aggregation rules and full evidence distribution; Appendix E links diagnoses to their sources.

Execution support in the successful pool includes probes actually executed within the trajectories and the Kysely repair replay. The five local repair replays contribute part of the execution evidence; propagation omissions in the failed pool may also coexist with other errors.

Task concentration and scope sensitivity. The five tasks with the most terminal annotations account for $197/473 = 41.6\%$ of the annotated runs. Relations are deduplicated by task, producer-side semantic change, adaptation obligation, and existing consumer path; a single run may involve multiple relations. Repeated runs describe recurrence of these relations; neither run counts nor relation counts represent independent samples. In the manual scope review, all terminal events in 290 runs were undisputed, 26 runs contained both disputed and undisputed events, and another 157 runs contained only disputed events. The first two groups cover 30 tasks, with 82 runs passing the original evaluation and 234 failing it; disputed and undisputed events in mixed runs are recorded separately. Appendix B provides the scope groups, evidence types, and counting conventions.

4.2 Recovery and Terminal Omissions Can Coexist

370 runs contain 415 recovery events across 36 tasks; 83 of these runs still have other terminal annotations (Table 2(b)). Recovery and terminal omissions are counted from their respective event records; the remaining 2,119 runs have neither event annotation. Together, these statistics charac-

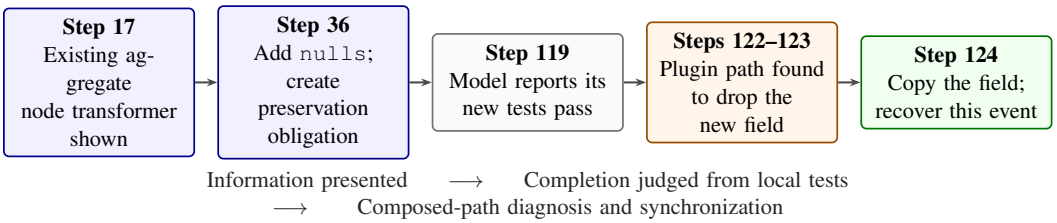


Figure 2: Natural recovery in Kysely `5WQPAYr`. Steps are assistant steps numbered from 1; horizontal spacing indicates order only. Step 119 records the model's statement about its own tests; subsequent actual diagnosis localized the gap in the plugin path. This run is independent of the terminal record in Figure 1.

terize the intermediate and final states of change propagation, while natural recovery trajectories provide additional process evidence on when gaps were discovered and how synchronization was performed.

4.3 Other Errors Define the Diagnostic Boundaries

Among the primary labels in the pool that failed the original evaluation, specification/interface errors account for 642 runs and local algorithmic errors for 418, together comprising $1{,}060/1{,}593 = 66.5\%$; the remaining labels are terminal propagation omissions (327), evaluation-process anomalies (48), environment issues (47), unlocalized compatibility issues (14), other unresolved cases (85), and incomplete runs (12). Primary diagnoses follow the protocol's priority order, with coexisting diagnoses recorded separately. The complete success/failure cross-tabulation is provided in Appendix B.1.

We diagnose interface and algorithmic errors from their specific behavior and trace missing tests, timeouts, and parent-test failures through the logs. Build gates, test aggregation, and program failures are assessed separately. We identify propagation omissions through an edit–obligation–consumer–diagnosis relation.

5 Propagation Paths and Discovery Processes

5.1 Terminal Omission and Delayed Recovery of the Same Type of Gap

In Kysely, `wq82S5C` (`inject-file`) incorporated `nulls` into the node definition, construction interface, and SQL compilation, but omitted the aggregate-node branch of the existing `OperationNodeTransformer`. This branch still copied the old fields one by one, allowing it to produce a type-valid new node that lost user-specified semantics. This terminal case records a program-level propagation gap under a condition in which a repository index was provided.

Another `inject-file` run on the same task, `5WQPAYr`, exhibits a traceable recovery process (Figure 2). Step 17 displayed the old aggregate transformation function in full, and step 36 added the new field; at step 119, the model reported that its newly added tests passed. Only at steps 122–123 did it discover that plugin transformation dropped the field, and step 124 added the missing copy. This independent run provides the complete temporal sequence of code reading, modification, gap discovery, and recovery through synchronization.

5.2 A Shared Entry Point Does Not Guarantee a Shared Consumer Path

In Query, `MYCfNHF` (`bare`) introduced a return wrapper carrying recovery state and unwrapped it in the internal `fetchFn` for ordinary queries. Infinite queries, however, replaced `context.fetchFn` through a behavior extension, bypassing this logic. At step 130, the model inferred identical recovery behavior because both used the outer `Query.fetch`, but a shared outer entry point does not guarantee that internal results pass through the same consumer. The original record reports $7/8$ new tests and $42/42$ existing tests passing, with the failed test targeting the re-

covery state of infinite queries. This case grounds the abstract notion of a "dependency omission" in a consumer operation bypassed by a replaced branch.

In termenv, `5FGCqii` (`bare`) passed `preserveResets` from the output object to style objects and templates, but ordinary rendering retained the old concatenation logic, leaving the new state consumed only on the truncation path. The tool output requested at step 4 displayed the old `Styled` in full; at steps 32 and 56, the model still explicitly concluded that the state affected only truncation. Once the consumer began responding to the state, the truncator's existing logic for reopening styles also required coordination. These two cases show partial integration of a return representation and a rendering state, respectively, and demonstrate that synchronization obligations must be checked along actual execution paths rather than inferred from a shared object or function name.

5.3 Explicit Awareness of a Gap May Still Leave Synchronization Incomplete

In Helm, `BRvPrLE` (`bare`) added merge strategies for arrays, but passed only the command-line strategy when reusing configuration during an upgrade, omitting Chart annotations. At step 135, the model explicitly discussed this gap, then concluded that the subsequent merge stage would compensate for it. In fact, the subsequent stage could not recover array elements already discarded by the preceding stage. This record contains an explicit discussion of the gap, an incorrect judgment about a subsequent stage, and a final program state that remained unsynchronized.

6 Execution Checks of Consumer Relations and Cross-Benchmark Counterexamples

6.1 Fixed Patches, Fixed Checks, and Local Synchronization

We selected five locally executable DeepSWE cases covering node fields, return wrappers, rendering state, configuration sources, and error-value representations. For each, we fixed the repository starting point, original model patch, and check program, then compared the original patch with local consumer synchronization. Repairs followed task behavior and consumer paths without adopting the reference patch.

Table 3 reports the actual execution results. Each row corresponds to one run, with multiple assertions covering different paths; parent and child pass events in the original suite are reported according to the test records. Program-level confirmation is based on restoration of the target behavior under fixed inputs and preservation of the listed controls, linked to specific terminal events. The complete repair process and execution scope are provided in Appendix C.

6.2 Repairable Semantic Loss Despite Passing All Tests

The Kysely replay applied the original patch to the fixed starting point `91cf3733`, constructed queries through the public API, and inspected AST attributes and SQL compilation output. Both `ignoreNulls()` and `respectNulls()` retained their semantics on the direct path; after passing through the new `SimplifyFramePlugin`, the generic node transformer, or the existing `WithSchemaPlugin`, both modifiers disappeared. For example, `first_value(...) ignore nulls over(...)` became `first_value(...) over(...)`. Adding one line to copy the field restored all six transformation checks, while both direct-path controls continued to pass.

The benchmark validator applied `test.patch`; the original evaluation comprised 254 new acceptance tests (F2P) and 22 existing regression tests (P2P). The record of these 276 passing tests, together with the additional checks, localizes a blind spot on composed paths; these tests are counted separately from the agent's own tests and our probes. In supplementary reruns, both the original patch and the one-line repair passed the same 276 original evaluation tests, as well as type checks for the production source in ESM/CJS and the Node test code (Appendix C). These results confirm a semantic gap in composition within this successful patch that can be repaired by synchronizing a single field in one line.

Table 3: Local program replays of five fixed patches. The target column reports the number of passing checks in the same set before and after repair; the preserved column lists checks that pass in both versions. Each row represents one trial; individual checks are not treated as independent samples.

Task/trial	Local synchronization	Target passes	Preserved passing checks
Kysely `wq82S5C`	Transformer copies the `nulls` field (1 file, 1 added line)	$0/6 \rightarrow 6/6$	$2/2$ controls; original evaluation $276/276$; source and test type checks
Query `MYCfNHF`	Shared result entry point unwraps the restoration marker (1 file, net 4 added lines)	$0/4 \rightarrow 4/4$	$8/8$ ordinary restoration and fresh-request controls
termenv `5FGCqii`	Renderer reads preservation state; truncation avoids reopening styles twice (2 files)	$0/7 \rightarrow 7/7$	$12/12$ controls; original repository tests
Helm `BRvPrLE`	Configuration merge reads annotation policies and retains CLI precedence (2 files)	$0/5 \rightarrow 5/5$	$48/48$ controls and regression leaf tests
Expr `7uAWePB`	`string()` preserves error values and reads their messages (2 files)	$0/6 \rightarrow 6/6$	$9/9$ controls; original repository tests

Kysely belongs to `inject-file`, and the other four trials belong to `bare`; these groups indicate the index-provision condition, while visibility of specific code is checked against each trajectory. Helm's 48 checks comprise 9 local controls, 5 existing action regression leaf tests, and 34 util leaf tests. Original evaluation outcomes and local replays are recorded separately; Appendix C details the full execution scope and changes in non-target behavior. The final Query repair includes explicit generic type annotations; Table 10 reports the original evaluation reruns.

6.3 Repair Effects and Boundaries Across Propagation Relations

For Query, unwrapping at the shared result entry point restored the infinite-query target while preserving the ordinary-query restoration and fresh-request controls. For termenv, coordinating the renderer and truncator allowed the preservation state to be consumed while avoiding duplicate style reopening. For Helm, incorporating the annotation strategy when reusing configuration preserved CLI precedence. In Expr, `7uAWePB` (`bare`) bound an error pointer to the catch variable, but the old `string` path dereferenced it and lost the `Error()` method. Preserving the error value restored message access, while also changing the string representation of error values supplied through the environment. Counts for each case appear in Table 3; Appendix C lists the specific execution scope, changes to non-target behavior, residual failures, and repair-development records. Supplementary reruns of the four original evaluations introduced no new test failures, and the final Query repair also passed execution and type checks for the four relevant packages.

6.4 Failure Diagnosis and Successful-Patch Counterexamples on SWE-bench Verified

We ran and saved 500 final SWE-bench Verified trajectories spanning 12 repositories, using the same `deepseek-v4-flash` model and SWE-agent 1.1.0. The patches in the trajectories, predictions, and original evaluation were all byte-for-byte identical; the original evaluation outcomes were 462 passes and 38 failures. Archived settings and retry details appear in Appendix F.

Evidence-based diagnosis of each of the 38 records that failed the original evaluation yielded 17 incomplete repairs, 11 mismatches with specification/interface or representation requirements, 7 local algorithmic errors, 2 human-confirmed terminal propagation omissions, and 1 evaluation anomaly. For Sphinx 9461, synchronizing only three lines in the consumer made all 62 original evaluation tests pass; two other runs contained recovery events. This failed subset is counted separately from DeepSWE; its categories and boundaries appear in Table 12.

We purposively selected three cases from the successful pool for execution replays of the original patches: the old `:any:` path failed to recognize new entry keys in Sphinx 7440; the old product reduction discarded the delta range in SymPy 12419; and variable attributes incorrectly entered the

coordinate-merging path in xarray 6461. Local synchronization restored the targets in all three cases, while the executed controls related to the original tasks continued to pass; the three-stage execution and regression scope are provided in Appendix F.

Terminal and recovery cases on FeatureBench. We analyze only the 46 completed (`Submitted`) answers from 100 Qwen3.8-Flash tasks; the other 54 timed out (`TimeExceeded`) and are excluded. Submission is separate from evaluation: 9 resolved and 37 failed. We manually confirmed three terminal diagnoses involving dependency naming, host patterns, and plotting styles; two have static consumer-path evidence and one has execution support. Four other runs contain five execution-supported recovery events, all repaired by the original agent before submission. These cases extend the evidence to another model and benchmark using the same `mini-swe-agent 2.4.6` framework (Appendix G).

7 Discussion and Limitations

Task evaluation and change impact. Execution counterexamples from DeepSWE and SWE-bench Verified reveal consumer defects undetected by the original task evaluations. This motivates checking consumers' new obligations and preserving evaluation outcomes, terminal states, and recovery records separately.

Program-level and cognitive attribution. Local repair checks examine the effects of code on behavior. Our analysis of natural recovery and the original conditions is descriptive. Causal effects of information refresh, repository indexes, code graphs, and retrieval strategies require controlled interventions, which fall outside this study; cognitive mechanisms also require separate evidence.

Measurement boundaries and generalization. Benchmark studies examine potential model-memory confounds and compare agent harnesses under controlled evaluation protocols (Prathifkumar et al., 2025; Zheng et al., 2026). We conduct a descriptive audit of the fixed corpus, reporting evidence strata, scope groups, run counts, and task coverage separately, with exploratory groups in Appendix B. DeepSWE and SWE-bench Verified use the same model while tasks, budgets, and frameworks vary together. FeatureBench adds Qwen terminal and recovery cases under different task and run conditions in the same framework as DeepSWE, without supporting model rankings or mismatch-rate comparisons. We selected replay cases for localizability and executability; conclusions concern the tested inputs, consumer repairs, and listed regression checks. Overall mismatch rates, average repair benefits, and causal effects across frameworks are outside the scope of our estimates.

8 Conclusion

Edit-induced dependency mismatch can cause evaluation failures and can also leave consumer defects in patches that pass the evaluation. The audit established terminal diagnoses in 473 runs including disputed-scope cases; 290 have all terminal events undisputed, and 158 of these have execution support for at least one terminal event. Targeted replays on two benchmarks examined the behavioral effects of local synchronization and exposed propagation defects missed by the original task evaluations, while Qwen cases on FeatureBench add cross-model evidence of terminal mismatches and recovery. These findings support separate measurement of task evaluation outcomes, terminal states, and recovery, without estimating broader mismatch rates.

AI USAGE STATEMENT

No AI was used in this work.

ETHICS STATEMENT

Not applicable.

REPRODUCIBILITY STATEMENT

The supplementary materials provide all DeepSWE model responses and scores, together with final patches, audit labels, and terminal diagnostic opinions, enabling reproduction of the results reported in this paper.

REFERENCES

Aider contributors. Repository map and map refresh configuration. https://aider.chat/docs/repomap.html, 2026.

Antonis Antoniades, Albert Örwall, Kexun Zhang, Yuxi Xie, Anirudh Goyal, and William Wang. Swe-search: Enhancing software agents with monte carlo tree search and iterative refinement. In Y. Yue, A. Garg, N. Peng, F. Sha, and R. Yu (eds.), *International Conference on Learning Representations*, volume 2025, pp. 64485–64515, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/a1e6783e4d739196cad3336f12d402bf-Paper-Conference.pdf.

Ramakrishna Bairi, Atharv Sonwane, Aditya Kanade, Vageesh D. C., Arun Iyer, Suresh Parthasarathy, Sriram Rajamani, B. Ashok, and Shashank Shet. Codeplan: Repository-level coding using llms and planning. *Proceedings of the ACM on Software Engineering*, 1(FSE):675–698, July 2024. ISSN 2994-970X. doi: 10.1145/3643757. URL http://dx.doi.org/10.1145/3643757.

Shawn Bohner and Robert S. Arnold. *Software change impact analysis*. IEEE Computer Society Press, 1996. URL https://ci.nii.ac.jp/ncid/BA29120744.

Dong Chen, Shaoxin Lin, Muhan Zeng, Daoguang Zan, Jian-Gang Wang, Anton Cheshkov, Jun Sun, Hao Yu, Guoliang Dong, Artem Aliev, Jie Wang, Xiao Cheng, Guangtai Liang, Yuchi Ma, Pan Bian, Tao Xie, and Qianxiang Wang. Coder: Issue resolving with multi-agent and task graphs, 2024. URL https://arxiv.org/abs/2406.01304.

Dhruv Gautam, Spandan Garg, Jinu Jang, Neel Sundaresan, and Roshanak Zilouchian Moghaddam. Refactorbench: Evaluating stateful reasoning in language agents through code. In *International Conference on Learning Representations*, March 2025. URL https://www.microsoft.com/en-us/research/publication/refactorbench-evaluating-stateful-reasoning-in-language-agents-through-code/.

Deyao Hong, Yizhe Chi, Wenyi Li, Xiaoqiu Wang, Mingju Gao, Kaisen Yang, Bingxiang He, Youjie Zheng, Calvin Xiao, and Qinhuai Na. Swe refactor bench: Can coding agents complete a long-horizon, whole-repository stack migration?, 2026. URL https://arxiv.org/abs/2608.23564.

Wenqi Huang, Charley Lee, Leonard Tng, and Serena Ge. Deepswe: Measuring frontier coding agents on original, long-horizon engineering tasks, 2026a. URL https://github.com/datacurve-ai/deep-swe.

Wenqi Huang, Charley Lee, Leonard Tng, and Serena Ge. Deepswe: Measuring frontier coding agents on original, long-horizon engineering tasks, 2026b. URL https://arxiv.org/abs/2607.07946.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? In B. Kim, Y. Yue, S. Chaudhuri, K. Fragkiadaki, M. Khan, and Y. Sun (eds.), *International Conference on Learning Representations*, volume 2024, pp. 54107–54157, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/file/edac78c3e300629acfe6cbe9ca88fb84-Paper-Conference.pdf.

Junwei Liu, Kaixin Wang, Yixuan Chen, Xin Peng, Zhenpeng Chen, Lingming Zhang, and Yiling Lou. Large language model-based agents for software engineering: A survey, 2025. URL https://arxiv.org/abs/2409.02977.

Oorja Majgaonkar, Zhiwei Fei, Xiang Li, Federica Sarro, and He Ye. Understanding code agent behaviour: An empirical study of success and failure trajectories, 2025. URL https://arxiv.org/abs/2511.00197.

Rahul Nanda, Chandra Maddila, Smriti Jha, Euna Mehnaz Khan, Matteo Paltenghi, and Satish Chandra. Wink: Recovering from misbehaviors in coding agents, 2026. URL https://arxiv.org/abs/2602.17037.

Siru Ouyang, Wenhao Yu, Kaixin Ma, Zilin Xiao, Zhihan Zhang, Mengzhao Jia, Jiawei Han, Hongming Zhang, and Dong Yu. Repograph: Enhancing ai software engineering with repository-level code graph. In Y. Yue, A. Garg, N. Peng, F. Sha, and R. Yu (eds.), *International Conference on Learning Representations*, volume 2025, pp. 30098–30121, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/4a4a3c197deac042461c677219efd36c-Paper-Conference.pdf.

Thanosan Prathifkumar, Noble Saji Mathews, and Meiyappan Nagappan. Does swe-bench-verified test agent ability or model memory?, 2025. URL https://arxiv.org/abs/2512.10218.

Yuqiao Tan, Jinxiang Meng, Fangyu Lei, Minzheng Wang, Shizhu He, Jun Zhao, and Kang Liu. Swe-touch: Benchmarking coding agents when users touch the code, 2026. URL https://arxiv.org/abs/2608.02499.

Wei Tao, Yucheng Zhou, Yanlin Wang, Wenqiang Zhang, Hongyu Zhang, and Yu Cheng. Magis: Llm-based multi-agent framework for github issue resolution. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (eds.), *Advances in Neural Information Processing Systems*, volume 37, pp. 51963–51993. Curran Associates, Inc., 2024. doi: 10.52202/079017-1647. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/5d1f02132ef51602adf07000ca5b6138-Paper-Conference.pdf.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Daniel Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. Openhands: An open platform for ai software developers as generalist agents. In Y. Yue, A. Garg, N. Peng, F. Sha, and R. Yu (eds.), *International Conference on Learning Representations*, volume 2025, pp. 65882–65919, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/a4b6ad6b48850c0c331d1259fc66a69c-Paper-Conference.pdf.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Demystifying llm-based software engineering agents. *Proceedings of the ACM on Software Engineering*, 2(FSE):801–824, June 2025. ISSN 2994-970X. doi: 10.1145/3715754. URL http://dx.doi.org/10.1145/3715754.

John Yang, Carlos Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (eds.), *Advances in Neural Information Processing Systems*, volume 37, pp. 50528–50652. Curran Associates, Inc., 2024. doi: 10.52202/079017-1601. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/5a7c947568c1b1328ccc5230172e1e7c-Paper-Conference.pdf.

Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. RepoCoder: Repository-level code completion through iterative retrieval and generation. In Houda Bouamor, Juan Pino, and Kalika Bali (eds.), *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 2471–2484, Singapore, December 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.emnlp-main.151. URL https://aclanthology.org/2023.emnlp-main.151/.

Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. Autocoderover: Autonomous program improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '24, pp. 1592–1604. ACM, September 2024. doi: 10.1145/3650212.3680384. URL http://dx.doi.org/10.1145/3650212.3680384.

Mengyu Zheng, Kai Han, Boxun Li, Haiyang Xu, Yuchuan Tian, Wei He, Hang Zhou, Jianyuan Guo, Hailin Hu, Lin Ma, Chao Xu, Guohao Dai, Lixue Xia, Yunchao Wei, Yunhe Wang, and Yu Wang. Claw-swe-bench: A benchmark for evaluating openclaw-style agent harnesses on coding tasks, 2026. URL https://arxiv.org/abs/2606.12344.

Qixing Zhou, Jiacheng Zhang, Haiyang Wang, Rui Hao, Jiahe Wang, Minghao Han, Yuxue Yang, Shuzhe Wu, Feiyang Pan, Lue Fan, et al. Featurebench: Benchmarking agentic coding for complex feature development. *arXiv preprint arXiv:2602.10975*, 2026.

A Audit Criteria and Version History

A.1 Data Freeze and Provenance

The earlier audit used the `deepswe-retrospective-census-v1.0` protocol. Earlier exploration examined evaluation outcomes in the original failed pool and inspected failure signatures across 24 tasks and 896 failed records; these counts describe the exploration workload. The archival source for the final statistics is `full_audit/canonical/primary.jsonl`; its study-archive manifest confirms complete coverage of all 2,879 runs and successful structural validation. The submission includes a minimal cohort projection at `deepswe_core_evidence_slim/data/final_labels.jsonl`, together with the original scores and patches. Current terminal-event evidence and scope labels are supplied separately in `deepswe_473_terminal_evidence/audit_opinions.jsonl`. Coverage of event-level diagnostic fields is reported separately in Appendix B. The archival source file's SHA-256 hash is

```
5479ebf42555db8b59e8f441285f8b937
becdcbd5ea4d315f7ab9cbfaa060877
```

The hash and structural validation verify data provenance and version consistency.

The human annotation guidelines were preregistered before exploratory analysis. After exploration and refinement of the operational criteria, three human reviewers independently re-reviewed all runs under common criteria to determine the classifications and evidence assignments used in this paper. A separate independently sampled blinded audit provided additional validation alongside the full-corpus blinded review. We determine diagnoses through the full human review; the submitted core directory retains run identifiers and source hashes for linking the distributed records. In the study archive, each audit record links the task, patch, relevant trajectory windows, and available diagnostics, and preserves alternative explanations and limitations of the materials. Aggregate statistics deduplicate runs by their unique identifiers.

A.2 Unified Classification Boundaries

Table 4 organizes the retrospective diagnostic criteria applied to the frozen audit into a sequence of checks against the source materials. Each decision concerns a specific propagation relation; a run's primary category incorporates coexisting diagnoses according to the protocol's priority order.

Conflicting obligations and the threshold for static evidence. We identify the specific behaviors supported by the task wording and the existing contract separately, and check their respective scopes of applicability. The reference patch serves only as an interpretive aid; when conflicting interpretations prevent the expected behavior from being determined, the obligation remains unresolved. If the obligation and a reachable deviation have been established, but whether the task description covers the input combination remains disputed, we retain the diagnosis and record a scope dispute. A static derivation must specify the entry point, input/state, branch conditions, call/data flow, and erroneous outcome, and check for guards that block the path or branches that are not connected; functions with the same name, changes across multiple files, or failures in other runs are insufficient substitutes.

Boundary examples. The field loss in Kysely `wq82S5C` meets the inclusion criteria and is supported by a repair replay; the combination of configuration reinjection and append in Helm `27jewNv` exhibits a statically reachable deviation, but whether the rendering combination falls within the task scope is recorded as a separate dispute. In Go-genai `JupuHoi`, the retroactive modification of response snapshots occurs within the new accumulator itself; despite an actual failure, it is classified as a local algorithm error rather than propagation. In Eicrud `5Xa3tcN`, the existing client's subsequent retrieval combines offset with the new cursor restriction, creating a conflict, but it could not be established whether the server-side `$find` task imposes this obligation for the client's new usage, so propagation remains unresolved. The latter two cases are excluded from the 473 runs; their primary labels are retained in the submitted cohort projection.

Primary labels organize records according to the protocol's priority order, while secondary labels and events preserve coexisting diagnoses. Program evidence and process evidence are used sepa-

Table 4: Unified decision table for propagation relations. Scope status and evidence level are recorded separately; confirmed exclusions and insufficient materials are treated separately.

Criterion	Required evidence	Treatment when false or indeterminate
Implemented edit	Baseline, patch, and trajectory from the same run, identifying the actual change Δ	Plans alone are excluded; missing materials remain recorded as missing
Established obligation	Exact task wording or a still-applicable existing contract explaining why the consumer must adapt	Differences from the reference implementation are insufficient; unresolved conflicts in obligations remain unresolved
Existing and reachable	An independent consumer present before the patch; a specific entry point, input/state, branch conditions, and consumption chain	Errors within a newly written/rewritten function are classified separately; confirmed unreachability leads to exclusion, while gaps in the path remain unresolved
Substantiated diagnosis	An actual erroneous output for the same relation, or a specific deviation derivable from source code	Evaluation outcomes, test names, and self-reports are insufficient substitutes; cases without a verifiable diagnosis remain unresolved
Terminal state/recovery	A gap remaining in the final patch; or a runtime failure of an existing consumer that is subsequently repaired	Repaired gaps are not counted as terminal; migration transients and generated-file synchronization do not automatically count as recovery
Scope and aggregation	Scope disputes recorded separately after the obligation and diagnosis are established, followed by separate counts of events and runs	Scope status follows explicit human adjudication; historical missing fields and supplementary decisions are recorded separately

rately: execution-evidence levels follow the recorded behavior, and model-visible code is identified in the messages presented to the model. Complete archived outputs support program analysis; unresolved relations with insufficient evidence and relations excluded on sufficient evidence are recorded separately. Our process analysis focuses on the focal cases, linking code presentation, explicit judgments, and synchronization actions to their trajectory locations.

A.3 Human Review Procedure

The annotation guidelines were preregistered before exploratory analysis. Full-corpus blinded human review followed refinement of the operational criteria. Three reviewers each independently reviewed all $2{,}879$ runs under the unified criteria and blinded conditions, covering runs that passed and failed the original evaluation. For each run, reviewers checked the task requirements, final patch, execution records, and relevant trajectories, and recorded the classification, consumer obligation, diagnostic basis, and alternative explanations. The full-corpus blinded review hid the original task-evaluation score summaries, existing audit labels, and the other reviewers' labels and rationales while retaining the program evidence needed for diagnosis. Before discussion in this full-corpus blinded stage, unanimous agreement among all three reviewers on the primary category was 97%, with all $2{,}879$ runs as the denominator; unanimous agreement on the complete event tuple $e = (\Delta, O, C, D)$ before discussion was 89%, with event comparisons covering both terminal and recovery events. The former measures agreement on the run's primary category; the latter requires agreement on all four components of the complete tuple. Evidence rechecking and discussion resolved all reviewer disagreements; the remaining "unresolved" labels denote insufficient diagnostic evidence. We report both pre-adjudication unanimous agreement rates at the aggregate level as integer percentages.

Independently sampled blinded audit. We conducted a separate independently sampled validation in addition to the blinded review of all $2{,}879$ runs. This stage hid the original task-evaluation score summaries and existing audit labels while retaining task requirements, source code, original model patches, and relevant execution evidence. Reviewers first made independent judgments and fixed their initial annotations, followed by unblinded comparison and adjudication of disagreements. The 97% and 89% figures above describe agreement in the full-corpus blinded review; the independently sampled audit is a separate procedure. This subsection describes the procedure; the submitted files contain the final cohort labels and terminal diagnosis opinions.

B Full Classification, Task Distribution, and Sensitivity

B.1 Primary Labels and Original Evaluation Outcomes

Table 5: Complete distribution of primary labels in the final audit. Terminal diagnoses include disputed-scope cases. Each trial receives one primary label; coexisting diagnoses and recovery events are recorded separately. "No failure diagnosed" denotes runs for which available evidence did not locate a specific failure.

Primary label	Original outcome: passed	Original outcome: failed	Total
Terminal propagation diagnosis	146	327	473
Propagation omission recovered within the trajectory	133	0	133
Specification/interface mismatch	39	642	681
Local algorithm or boundary error	67	418	485
Compatibility regression; propagation chain not located	9	14	23
Environment/build	0	47	47
Evaluation process anomaly	0	48	48
Incomplete/empty patch	0	12	12
No failure diagnosed	871	0	871
Unresolved	21	85	106
Total	1,286	1,593	2,879

The 133 runs whose primary label is recovered propagation are fewer than the 370 runs containing recovery events: the remaining runs may also have terminal or other diagnoses with higher priority. "Evaluation-process anomaly" includes build conflicts caused by overlaying hidden tests or problems with evaluation mapping; the corresponding patch diagnoses are recorded separately. "Unlocalized compatibility issue" records an observed regression whose propagation relation remains unresolved.

B.2 Explicit Scope Disputes and Evidence Types

From event records to aggregate statistics. For each diagnosed event j in run r whose terminal or recovery state is established, we record $L_{rj} \in \{\text{terminal}, \text{recovered}\}$ alongside the original tuple $e = (\Delta, O, C, D)$. For the terminal-event strata reported here, we use the additional labels $E_{rj} \in \{\text{exec}, \text{static}\}$ and $S_{rj} \in \{\text{undisputed}, \text{disputed}\}$. The evidence labels are exclusive: execution-supported events have $E_{rj} = \text{exec}$; source-supported events lacking execution evidence have $E_{rj} = \text{static}$. These labels support aggregation; the 89% agreement statistic compares the four-part tuple e. For the $R = 2{,}879$ runs, define

$$T_r = \{j : L_{rj} = \text{terminal}\}, \qquad A_r = \mathbf{1}[T_r \neq \varnothing],$$
$$X_r = \mathbf{1}[\exists j \in T_r : E_{rj} = \text{exec}],$$
$$U_r = \mathbf{1}[T_r \neq \varnothing \ \wedge \ \forall j \in T_r : S_{rj} = \text{undisputed}].$$

Thus $\sum_r A_r = 473$, $\sum_r X_r = 177$, $\sum_r U_r = 290$, and $\sum_r U_r X_r = 158$. Since the current terminal evidence classification has 0 unresolved cases, $\sum_r A_r(1 - X_r) = 296$ counts runs whose terminal events are all statically supported. The $A_r = 0$ group comprises 2,222 not-identified runs and 184 terminal-status-unresolved runs, reported separately in Table 7; it records absence of an established terminal diagnosis, not verified absence of defects. Recovery events with $L_{rj} = \text{recovered}$ are counted separately; a run can contain both terminal and recovery events.

With task mapping $t(r)$, task coverage is

$$p_{\text{task}} = \frac{1}{64} \sum_{t=1}^{64} \mathbf{1}[\exists r : t(r) = t \ \wedge \ A_r = 1] = \frac{37}{64} = 57.8\%.$$

The event denominator is $N_{\text{event}} = \sum_r |T_r| = 546$, with $\sum_r \sum_{j \in T_r} \mathbf{1}[E_{rj} = \text{exec}] = 179$ execution-supported events and 367 statically supported events. Their shares, $179/546 = 32.8\%$

Table 6: Mutually exclusive run groups after manual scope review. Each run is grouped by the scope decisions for all of its terminal events; sources for the supplementary adjudication appear in the text.

Run group by scope	Original outcome: passed	Original outcome: failed	Total
All terminal events undisputed	75	215	290
Both undisputed and disputed events	7	19	26
All terminal events disputed	64	93	157
Total	146	327	473

and $367/546 = 67.2\%$, describe evidence composition within diagnosed terminal events. The 85 relation types are counted after deduplication within each task, as detailed in Table 8.

The 473 runs contain 546 terminal events. Our supplementary manual scope review classified all 334 events that previously lacked a separate scope field as undisputed; together with the original 6 undisputed events, this yields 340 undisputed events, while the remaining 206 events have scope disputes. The submitted `audit_opinions.jsonl` records each terminal event's final scope status by event identifier. We use these final manual scope decisions in the statistics.

Table 6 partitions runs into three mutually exclusive categories: 290 runs with all terminal events undisputed, 26 with both undisputed and disputed events, and 157 with all terminal events disputed. The first two categories total 316 runs and cover 340 undisputed events; the other, disputed events in the mixed category are excluded from the undisputed counts. Among disputed events, $15/191$ have execution/static support, respectively; among undisputed events, the corresponding counts are $164/176$.

Within the 290 runs whose terminal events are all undisputed, 158 have execution support for at least one terminal event: 5 originally passed and 153 failed. These 158 runs contain 175 terminal events, comprising 159 with execution support and 16 with static support; 16 runs contain both evidence types. Thus, inclusion in this third set requires undisputed scope for every terminal event and execution support for at least one event, while each event retains its own evidence level. These nested counts can be recomputed from the event labels and run groups in the submitted `audit_opinions.jsonl`.

Stratification by evidence source. All 546 terminal event diagnoses have undergone human review and a unified evidence-source classification: 179 have evidence of execution failure, 367 have static reachability evidence, and 0 have unresolved evidence. Each event records its `evidence_level`, decisive source, and an explanation linking the evidence to the final patch. Execution support comprises 155 test witnesses, 18 in-trajectory probes, 5 repair replays, and 1 direct compiler diagnostic. Execution support requires a recorded failure linked to the same consumer relation in the final patch. We assign static support to verifiable source-code derivations lacking execution evidence and record scope status separately from evidence sources.

At the run level, 177 runs contain at least one execution-supported terminal event and 296 contain only statically supported terminal events, with 0 evidence-unresolved runs; 29 runs contain both event types. Unresolved evidence here concerns the 473 runs and is distinct from the 184 runs with unresolved terminal status in the original full corpus. We aggregate statistics under this common evidence stratification. The submitted diagnosis opinions provide the event-level classifications, reasoning, final-patch bindings, and limitations. These classifications draw on the existing execution records, source code, and trajectories. The coverage of the original evidence supplied with this submission is specified in Appendix E.

B.3 Task Concentration

Tasks, relations, and runs. Table 8 reports 37 tasks, 85 classes of within-task consumer relations, 473 runs, and 546 terminal events side by side. Grouping uses the original diagnostic event as its unit: within a task, events with the same producer-side semantic change, obligation, and existing consumption path are grouped into one class, without counting differences in field names, implementation variants, evidence levels, or input values as separate classes; distinct obligations or

Table 7: Full evidence and terminal-status distribution. Any exec. requires at least one execution-supported terminal event; Static only requires all terminal events to be statically supported. The first three evidence columns partition the 473 diagnosed runs, including disputed-scope cases. Not identified and unresolved status retain their distinct diagnostic meanings.

Original evaluation outcome	Any exec.	Static only	Unresolved evidence	Not identified	Unresolved status	Total
Passed	11	135	0	1,087	53	1,286
Failed	166	161	0	1,135	131	1,593
Total	177	296	0	2,222	184	2,879

independent consumption paths remain separate. When an original event already records a tightly coupled consumption chain or an associated set of consumers, that granularity is retained instead of splitting it into multiple call-graph edges; each original event belongs to exactly one group, while specific consumer paths may overlap, and grouping boundaries are disclosed individually.

The 146 runs that passed the original evaluation cover 21 tasks and 37 relation classes; the 327 failed runs cover 35 tasks and 74 relation classes. The two pools share some tasks and relations, so the overall counts use their union. This section describes the distribution of diagnosed terminal relation types and their run coverage. The submitted `relation_opinions.jsonl` contains relation definitions, grouping rationales, member event and run identifiers, and counts.

The five tasks with the most terminal annotations are Helm array merging with 44 runs, termenv with 41, Tengo destructuring with 40, Scriggo with 38, and Expr with 34, totaling 197 runs, or 41.6% of all terminal annotations. The five tasks with the most recovery records account for $151/370 = 40.8\%$. We provide complete per-task counts linked to individual run diagnoses through stable task identifiers.

Of the 45 runs for the Helm array-merging task, 44 have terminal propagation annotations. Among them, 43 record the same repeated-concatenation mechanism, all supported by static reachability evidence and marked with task-scope disputes. These 43 records count as one class in the relation summary; all 49 terminal events for the task are grouped into 4 relation classes. Replay confirmation for the other run, `BRvPrLE`, concerns the annotation policy not being carried through to the configuration-reuse stage (Section 6); the repeated-concatenation relation retains its static evidence classification.

B.4 Previously Explored Tasks and the Remaining Tasks

Using the task list recorded during the earlier exploration, we partition the fixed population into the 24 tasks explored earlier and the remaining 40 tasks. Table 9 uses all runs in each group, including both successes and failures; the table reports the group distributions separately from the final event-level scope classifications.

The original failure share is 83.0% in the explored group and 38.7% in the remaining group, indicating that the earlier analysis concentrated substantially on difficult tasks. The shares with terminal annotations are 24.8% and 11.4%, respectively, describing the two groups under the common final audit criteria. The shares with unresolved primary labels are 4.4% and 3.3%, respectively, whereas the shares with unresolved terminal propagation status are 4.1% and 7.8%; these two unresolved rates measure uncertainty in the primary category and terminal propagation status, respectively. Explicit scope disputes mark boundaries of consumer obligations. Both groups underwent full human re-review under the same criteria; this partition describes historical exploration and is reported separately from the additional independently sampled blinded audit. The remaining 40 tasks are not a previously sealed, unseen holdout set. This table reports descriptive shares for the fixed corpus; relation aggregation captures repetition within tasks, while evidence sources and scope status are reported in separate strata.

Table 8: Within-task aggregation of relations in terminal diagnoses. Relation types are distinguished by the producer's semantic change, the adaptation obligation, and the existing consumer path; relations are not merged across tasks. A run may involve multiple relation types; runs per task are deduplicated using unique run identifiers.

Task identifier	Relation types	Runs	Events
abs-stepped-slices	1	10	10
anko-default-function-arguments	6	25	46
anko-typed-variable-bindings	3	12	12
arcane-drift-detection-baselines	1	21	21
arktype-json-schema-refs-depende	1	14	14
awilix-async-container-initializ	2	6	6
claude-code-by-agents-recursive	1	1	1
cliffy-config-file-parsing	1	4	4
drizzle-orm-window-function-buil	1	1	1
dynamodb-toolbox-conditional-att	3	11	14
dynamodb-toolbox-lazy-recursive	2	9	9
effect-sse-httpapi-streaming	1	1	1
expr-try-catch-errors	2	34	35
helm-array-merge-strategies	4	44	49
helm-unified-manifest-stream	3	6	7
ink-grid-box-layout	5	6	7
kea-atomic-signal-selectors	5	33	45
koota-composite-trait-aspects	4	14	14
koota-deferred-mutation-buffer	2	5	5
koota-pair-relation-tracking	3	10	11
koota-query-predicates	3	6	6
kysely-window-grouping-helpers	1	1	1
meriyah-explicit-resource-declar	2	2	2
ofetch-per-origin-circuit-breake	1	1	1
optique-conditional-option-depen	1	4	4
prometheus-transactional-reload	2	6	6
query-persist-restored-query-sta	1	1	1
quill-shared-toolbar-focus	2	22	25
scriggo-method-declarations	4	38	46
sql-formatter-bigquery-pipe-form	1	3	3
superjson-error-stack-serializat	2	8	8
tengo-callable-instance-isolatio	3	3	3
tengo-destructuring-bindings	5	40	57
termenv-preserve-ansi-resets	1	41	41
vitest-duration-sharding	1	2	2
yaegi-go-embed-directives	1	24	24
ytt-jsonpath-query-api	3	4	4
Total (37 tasks)	85	473	546

Table 9: Descriptive grouping of historically explored tasks and the remaining tasks. Percentages use the number of runs in each group as the denominator, and metrics can overlap. The scope row retains the explicit flags in the historical snapshot; Table 6 reports the groups after supplementary manual adjudication.

Metric	Exploratory group (24 tasks)	Remaining group (40 tasks)
Runs	1,079	1,800
Original evaluation passed	183 (17.0%)	1,103 (61.3%)
Original evaluation failed	896 (83.0%)	697 (38.7%)
Terminal diagnoses	268 (24.8%)	205 (11.4%)
Containing recovery events	144 (13.3%)	226 (12.6%)
Unresolved primary label	47 (4.4%)	59 (3.3%)
Unresolved terminal status	44 (4.1%)	140 (7.8%)
Containing explicit terminal scope disputes	108 (10.0%)	75 (4.2%)

B.5 Evaluation Outcomes and Program Diagnosis

Among the original 1,593 failed runs, 1,385 did not pass all new tests but passed all existing tests, 149 did not pass all tests in either group, and 59 passed all new tests but did not pass all existing tests. 213 runs contain signals of missing tests, including 166 in which every non-passing item is missing; 129 contain timeout signals. These latter two signals may overlap. These counts describe test-result signals and depend on the F2P/P2P partition, missing reports, and parent–child aggregation; semantic propagation events are diagnosed separately from specific program relations.

For example, a build gate can block multiple tests at once; a timeout in an existing test may stem from the environment or from a hang introduced by the edit. In some Obsidian records, both new and existing tests passed, but the evaluation mapping failed to recognize their names. These records establish specific evaluation false negatives. We retain the original scores and record diagnoses separately, linking them by run identifier for statistical analysis.

C Local Replay Setup and Validation Results

For each of the five replays, we preserve the repository starting point, original model patch, fixed checks, local repair diff, and execution outputs. The submitted core directory's `replays/register.json` links the five local replay kits to their run identifiers, diagnosed consumers, recorded results, and hashes. The five replayed model patches are byte-for-byte identical to their archived counterparts, and the result-file hashes match the linked records.

The supplementary regression runs use the original DeepSWE task tests and designated images; the content hashes of all four task directories match the original run lockfiles. Both versions use the same test lists, and new failures are compared individually by test identifier (Table 10). Table 10 and the case paragraphs below report the settings, repair versions, and results of these additional checks. The submitted replay kits correspond to their respective local-validation versions.

Table 10: Supplementary reruns of the original DeepSWE evaluations. F2P reports passing counts before and after repair; P2P reports the passing counts shared by both versions. New failures are counted relative to the original model patch.

Task	F2P: before → after	P2P: both	New failures
Kysely	254/254 → 254/254	22/22	0
Query	7/8 → 8/8	42/42	0
Helm	44/47 → 47/47	10/12	0
Expr	73/79 → 78/79	66,265/66,265	0

Kysely: `wq82S5C`. The replay executes AST transformations and SQL generation on the actual source code using Node's TypeScript transformation support. It checks two NULLS modifiers across

three transformation paths and retains two direct-compilation controls. In the submitted core directory, `cases/wq82S5C/verifier/test-stdout.txt.gz` records the verifier's cleanup and application of `test.patch`; the corresponding `ctrf.json.gz` records 254 passing F2P tests and 22 passing P2P tests. The supplementary rerun uses the original task tests and designated image: the original patch and the one-line repair both pass the same 276 tests, with no new failures. The fixed propagation checks yield $0/6 \rightarrow 6/6$ passes, and the direct-path controls remain $2/2 \rightarrow 2/2$. Under Node 24.12.0 and TypeScript 5.9.3, `tsc --noEmit` passes for the production ESM and CJS code and the Node test code in both versions. The semantic checks observe AST transformations and generated SQL; execution covers the tests selected by the original evaluation and the probes above. The submitted `replays/kysely/` kit reproduces the AST/SQL checks. Its type-check log records the initial environment, while the successful supplementary checks use the environment specified above.

Query: `MYCfNHF`. The replay runner uses esbuild 0.27.3 from the original lockfile and the original tsconfig, compiles 24 actual core modules, and fixes a set of 12 checks. Four infinite-query targets cover the combinations of synchronous/asynchronous persisters and clients/observers; eight controls cover ordinary restoration and new requests. Adding a net four lines of unwrapping logic at the shared result-consumption point restores the targets. The submitted `replays/query/` kit contains the fixed source, build dependencies, and local checks; an offline rerun in a separate directory has been completed. The supplementary original evaluation improves from $49/50$ to $50/50$. Runtime tests for the four packages—query-core, persist-client-core, React, and Preact—yield 1,502 passes and 1 pre-existing skip in both versions. The repair includes an explicit `TData` type annotation at the shared consumption point and passes the full type-checking matrix for all four packages. The annotation leaves the generated JavaScript unchanged. The submitted local kit uses `let data = ...`; the subsequent four-package validation uses `let data: TData = ...`. Both versions implement the same runtime unwrapping logic.

termenv: `5FGCqii`. The repair consumes the state in `style.go` and coordinates repeated style reopening in `ansi/truncate.go`, reusing the model's existing tokenizer. Both versions use the same color-environment normalization and test expectations. The original suite reports 122 passing parent-test and subtest events, covering 76 top-level tests. The separate zero-width OSC truncation target fails in both versions. Intermediate implementations and outputs are included in the core directory's `replays/termenv/development_initial/`.

Helm: `BRvPrLE`. The replay calls `Upgrade.reuseValues` in the actual action package, using the existing test infrastructure without requiring a Kubernetes cluster. The repair changes two source files, adding 17 lines and deleting 2: it adds a thin wrapper around the existing annotation parser and passes the policy to the configuration-reuse consumption stage while preserving CLI precedence. The 48 checks that remain passing in the table comprise 9 local controls, 5 existing action regression leaf tests, and 34 util leaf tests. In the supplementary original-evaluation rerun, three upgrade failures are resolved, two original lint failures remain, and no new failures occur. After retaining three fixture symbolic links in the original task image, additional tests cover five packages: action, util, two loaders, and the filesystem. Both versions pass $374/374$ deduplicated leaf tests, including six permission checks rerun as a non-root user. The submitted `replays/helm/` kit covers the five local targets and the 48 preserved checks; the additional original-evaluation and 374-test results are reported here. The full audit also includes a distinct duplicate-merge diagnosis that retains its static evidence classification; this replay concerns the annotation-source relation.

Expr: `7uAWePB`. With the original patch fixed, `string(e)` produces a struct representation enclosed in braces. After string conversion preserves the error value and uses its message semantics, all six targets and nine controls pass. The original repository suite yields 66,786 passing events and 1 skip both before and after repair; the suite counts parent-test and subtest events. The repair adds a net eight lines and also makes `string(E)` use the error-message representation for error values supplied through the environment, changing the output from {`env err`} to `env err`. In the supplementary original-evaluation rerun, P2P remains at 66,265/66,265 and F2P increases from $73/79$ to $78/79$, with no new failures. Using the unmodified task source as the behavioral baseline, additional compatibility checks preserve $18/22$ existing behaviors. All four changes involve non-nil pointer error values supplied through the environment switching to the message rep-

resentation; an additional $10/10$ catch-method and type controls remain passing. Thus, the repair restores the targets while extending some existing string-conversion semantics. The submitted `replays/expr/` kit contains the local checks and original repository-suite outputs; the additional compatibility and original-evaluation results are reported here. The original evaluation still has a failing `ErrtypeTypeError` test; the original evaluation and the six local targets are reported separately according to their check definitions.

D Positive and Negative Cases of Program Attribution and Process Evidence

Obligations for optional fields. A field that is optional in the type system may still carry semantics specified by a particular invocation. In Kysely, the input contains a NULLS modifier, which the consumer discards, thereby changing the output. This input–processing–output chain addresses path consistency more directly than type-checking results do.

Specifications and local implementations. If a requirement calls for a multiline report but the patch outputs a single line, the primary issue is specification implementation; an off-by-one index boundary or acceptance of an invalid numeric format is primarily a local algorithm error. We diagnose propagation by locating a distinct existing consumer's unmet adaptation obligation introduced by the agent's edit.

Transient inconsistency and recovery. Changing a definition before updating its callers is a normal editing sequence. Our audit counts recovery only when an existing consumer exhibits an observable runtime behavioral failure and is subsequently repaired. Compilation errors during an ongoing migration and synchronization of golden files do not count as such recovery; unresolved terminal compilation problems may instead be assessed through their specific propagation relations. These criteria address process events and final program states, respectively.

Information presentation and explicit judgments. Assistant steps are indexed from 1, serialized message positions from 0, and patch line numbers from 1. Visibility is determined by the tool-message content presented at the time; archived `raw_output` may retain middle sections omitted from the displayed tool output. The complete output supports program diagnosis; evidence of code presentation comes from the displayed messages. Step 17 of Kysely `5WQPAYr` supports code presentation, and step 135 of Helm `BRvPrLE` supports explicit discussion; these provide message-level process evidence. Step 130 of Query `MYCfNHF` corresponds to message 260 and records the inference based on a shared outer entry point. Step 4 of termenv `5FGCqii` corresponds to the read request in message 8, while message 11 presents the old renderer; steps 32 and 56 correspond to messages 72 and 120, respectively, and explicitly assert that `Styled` need not consume the state. These locations separately establish visible code and explicit judgments, distinguishing a tool request from its displayed response.

Evidence levels for additional cases. Quill `JMoRtHX` retains the previous editor's image-input restrictions when multiple editors share a toolbar. In Koota `BRMDKd8`, the edit stores `Not(aspect)` constraints in `forbiddenAspects`. The ordinary query evaluator handles this representation, but the existing `checkQueryTracking` omits the constraint when a tracking modifier is combined with `Not(aspect)`. This archived diagnosis has static reachability support, with task scope recorded as disputed. These cases concern preservation of associated state and adaptation of parallel query-evaluation paths, respectively. We also consider alternative process explanations involving code visibility, interface-design choices, and differences in specification interpretation. We report these cases according to their recorded evidence sources.

E Reviewable Artifacts and Source Mapping

Submitted directories and coverage. The submitted supplement is `deepswe_review_materials.zip`. After extraction, it contains two top-level directories: `deepswe_core_evidence_slim` (**C** in Table 11) and `deepswe_473_terminal_evidence` (**D**). Directory C contains all 2,879 original scores and final patches, 64 task descriptions, a minimal frozen-label projection,

nine selected visible interaction records with their archived verifier reports, and five local replay kits. Directory D contains readable and machine-readable diagnosis opinions for 473 runs and 546 terminal events, together with definitions and membership for 85 within-task relations.

Reading the current labels. In D, `audit_opinions.md` provides the readable entry point; `audit_opinions.csv` has one row per event, and `audit_opinions.jsonl` groups events by run. These records include the edit, obligation, existing consumer, diagnostic basis, final-patch binding, evidence level, scope status, and limitations. `relation_opinions.jsonl` provides the relation identifiers, grouping rationales, and member events. Use these current event and scope fields for Tables 2, 6, and 8; C retains the earlier frozen fields for provenance.

Table 11: File index for the submitted supplement. Paths are relative to directory C or D as defined above.

Content	Directory	Relative path
Original scores and patches	C	`data/scores.csv`; `patches/`
Task descriptions	C	`tasks/`
Frozen labels and census	C	`data/final_labels.jsonl`; `data/expected_counts.json`; `verify.py`
Nine selected interactions and verifier reports	C	`cases/index.json`; `cases/<trial>/visible_messages.json`; `cases/<trial>/verifier/`
Five local replay kits	C	`replays/register.json`; `replays/` under `kysely`, `query`, `termenv`, `helm`, `expr`
Current terminal opinions	D	`audit_opinions.md`; `audit_opinions.csv`; `audit_opinions.jsonl`
Relation definitions and members	D	`relation_opinions.jsonl`
Core-file integrity and provenance	C	`SHA256SUMS`; `data/provenance.jsonl`; `packaging/slimming_manifest.json`

Verification and evidence navigation. Run `python3 verify.py` in C to check its file hashes, score–label joins, frozen census, and local replay registrations. This script covers C; current terminal evidence, scope, and relation counts can be recomputed from D's opinion files. Run and event identifiers link the opinions to the distributed patches and selected case records. The visible interaction records cover the nine cases listed in `cases/index.json`; per-run hashes and response indices identify the corresponding study-archive records. The supplied replay runners reproduce the local checks within each kit's stated requirements. The review package focuses on DeepSWE cohort scores and patches, terminal diagnosis opinions, and five local replay kits; Table 11 specifies its contents. The corresponding appendices report the settings, results, and check scopes of the additional original-evaluation, type-checking, extended-regression, SWE-bench Verified, and FeatureBench analyses. The terminal opinions provide diagnostic reasoning and source coordinates for locating the associated study-archive records. Existing redactions and upstream code licenses are retained in the distributed material.

F Failure Diagnosis and Program-Level Counterexample Validation on SWE-bench Verified

Using our saved evaluation records, this section combines instance-level diagnosis of the subset that failed the original evaluation with replays of three successful patches to examine failure categories, recovery during the trajectory, and evaluation blind spots in another benchmark and coding framework. Archive verification establishes the correspondence among instances, configurations, patches, and evaluation outcomes; program replays identify specific propagation gaps and verify the effects of local synchronization.

F.1 Dataset, Configurations, and Evaluation Correspondence

The extended dataset comes from `baseline-full-trajectories.tar.xz` and contains one final saved trajectory for each of 500 instances, with 30,925 steps across 12 repositories and 499 distinct repository–base-commit combinations. Instance-by-instance comparison with a pinned version of the official Verified dataset confirms exact agreement in instances, repositories, base commits, and archived original problem statements.[1] The original archive preserves message histories, tool observations, and inputs at each step as the source trajectory records; a more readable copy omits duplicate nested copies of inputs.

All runs use `deepseek/deepseek-v4-flash`, SWE-agent 1.1.0, and SWE-ReX 1.4.0, with temperature 0 and top-p 1. For 411 runs, the call limit is 100 and the configured cost limit is 3; for 88 runs, these values are 100 and 0, respectively, and for the remaining 1 run, they are 200 and 3; 0 is reported solely as the original configuration value. The script permits retrying empty patches with a higher call limit and overwriting earlier outputs, so these 500 trajectories are the final saved trajectories, including retry results. The ratio $462/500$ is the archived pass rate for these trajectories, which differs in its counting protocol from first-attempt `pass@1` under a uniform budget. Some trajectories also consulted upstream issues, PRs, or newer implementations, with external repair materials accessible during task solving.

Both the per-instance `report.json` files and the aggregate table record 462 passing and 38 failing instances. All patches are nonempty and were applied successfully; `pylint-dev__pylint-4604` also carries an ambiguous `no_tests_collected` flag, so its failed evaluation status retains an explanation at the test-collection level. For every instance, the final patch in the trajectory, the prediction file, and `patch.diff` in the evaluation directory have identical SHA-256 hashes. We obtained these counts by executing the official evaluation and verified them against the saved per-instance reports; instance-level failure diagnoses and the three local replays below are reported separately.

F.2 The 38 Runs That Failed the Original Evaluation

We fix the diagnostic scope to the 38 runs marked `resolved=false` in the original archive, checking each problem statement, complete final patch, original evaluation log, and test patch, and tracing edits, failures, and subsequent corrections through the relevant trajectories. We reviewed the consumer obligations and classification boundaries and retained the supporting evidence. All 38 records and their evidence-based diagnoses have been individually checked and verified by human reviewers; Table 12 summarizes the results. The primary diagnosis explains why the original run ultimately failed the evaluation; recovery events are recorded separately.

Confirmed terminal propagation omission. In Sphinx 9461, autodoc newly generates the `:classmethod:` option, but the existing `PyProperty` parser has not been adapted; the original log explicitly reports `unknown option: classmethod`. On the pinned source with the complete original model patch applied, adding only three lines for the consumer's option definition and signature-prefix handling changes the same original evaluation suite from 61 passes and 1 failure to 62 passes; the corresponding unmodified source yields 59 passes and 3 failures. An actual HTML build also verifies restoration of documentation, types, registration, and references

[1] Official dataset: `https://huggingface.co/datasets/SWE-bench/SWE-bench_Verified`. The dataset snapshot and file hashes identify the version used in the study archive.

Table 12: Primary diagnoses for the 38 SWE-bench Verified runs that failed the original evaluation. Each run receives one terminal primary diagnosis; two recovery events are counted separately from the primary diagnoses.

Primary diagnosis	Runs	Evidence
Terminal propagation omission (human-confirmed)	2	Sphinx 9461 passes all 62 original evaluation tests after local synchronization; Astropy 13398's mismatch between new and existing coordinate-transformation paths is human-confirmed
Incomplete repair	17	Branches, inputs, or call paths required by the original task remain uncovered, including Django 11734
Specification/interface or representation mismatch	11	Implementation outputs differ from acceptance requirements, including strict representation requirements
Local algorithmic error	7	Algorithmic or conditional errors identified, without an independently established propagation relation
Evaluation-process anomaly	1	pylint 4604 fails during import; no tests are actually executed
Total	38	Originally failing subset, retaining original evaluation outcomes

for ordinary properties, abstract properties, class properties, and abstract class properties. The RST generated by autodoc is identical before and after the repair, preserving the added functionality and original assertions. This replay confirms the consumer relation under the listed checks. This case is reported separately from the three program-level counterexamples in the next subsection that passed the original evaluation.

Astropy 13398's new coordinate-transformation route produces numerical discrepancies in existing route-equivalence tests. Human review confirmed this relation as a terminal propagation omission based on the original execution records and review of the consumer obligation; Sphinx 9461 additionally has a local replay comparison before and after consumer synchronization.

Two recovery events. In Django 11087, narrowing the fields in deletion queries causes the existing `NestedObjects` display path to issue an additional deferred-loading query, increasing the original two queries to three; synchronizing field selection restores the count to two. This recovery in performance behavior is recorded separately from the final local algorithmic issue of still selecting the extra `origin_id` field. In scikit-learn 25102, passing the new `dtypes` argument to an existing override of `set_output` triggers a `TypeError`; maintaining compatibility with the old call restores the same test group. This approach is later withdrawn, and the final type-preservation requirement remains unmet. The two runs receive primary diagnoses of local algorithmic error and incomplete repair, respectively, with recovery events recorded separately.

Classification and evaluation boundaries. Additional validation of Django 11734 shows that the original model patch combined with the consumer repair restores the original query test group, but another explicit nested-reference path in the unmodified source also exhibits the same kind of stringification. The boundary between a new consumer obligation and an incompletely repaired pre-existing consumer defect is insufficiently clear, so we conservatively classify this run as an incomplete repair and retain it in the set. In Django 16263, a local optimization algorithm error in annotation pruning breaks count/grouping equivalence. Removing and then restoring the `None` guard in the same function in pylint 4604 is an ordinary local correction; its final evaluation fails to collect tests because the test code imports the nonexistent `IS_PYPY`. A local check of SymPy 21596 produces `FiniteSet(1/2)`, whereas the original evaluation requires a mathematically equivalent unevaluated set; this case is reported as a difference in representation requirements. Some runs' evaluation reports contain truncated test identifiers or differences in assigned names; diagnosis uses the actual failure logs for localization, and original evaluation outcomes are retained. This table uses

the fixed historical subset of 38 originally failing runs; subsequent problem-statement corrections and reevaluations (see Section F.3) are recorded separately.

F.3 Three Program-Level Counterexamples That Passed the Original Evaluation

For each of the three cases, we apply the complete original model patch to its pinned starting revision and then compare it with a locally synchronized version; program inputs and expected correctness remain consistent with the original task's repair requirements. The original evaluation columns count test identifiers in our saved official evaluation reports; the three cases are purposefully selected program-level counterexamples for validation through execution.

Table 13: Three supplementary SWE-bench Verified replays. All passed the original evaluation; synchronization repairs confirm only the listed consumer relations.

Instance	Original F2P/P2P	Diagnosis with the complete model patch	After local synchronization
Sphinx 7440	1/1; 12/12	`:any:` `` `MySQL` `` cannot find the case-preserving glossary entry	Reference and HTML restored; the original repair allowing differently cased entries to coexist is preserved
SymPy 12419	1/1; 25/25	For a range-constrained delta in a product, reduction before substitution yields 1, whereas direct substitution yields 0	Both orders yield 0; controls for the old two-argument form, in-range cases, empty products, and the original task are preserved
xarray 6461	1/1; 247/247	`where(..., keep_attrs=True)` overwrites coordinate units in seconds with the variable's kelvin units	Coordinate attributes restored; the original task's scalar-input repair is preserved

Sphinx: glossary keys and the general-reference consumer. The edit in `sphinx-doc__sphinx-7440` preserves the case of glossary keys and adapts the dedicated `:term:` path, but misses the old `resolve_any_xref` lookup, which lowercases glossary entries uniformly. Building a minimal HTML document with the actual source shows that the previously working `:any:` `` `MySQL` `` reference consequently fails to resolve. Reusing the adapted glossary resolver in the glossary branch of general-reference resolution restores the reference. The original functionality fixture requires differently cased entries to coexist: strict-build exit codes at the original starting revision, with the model patch, and with the synchronized version are $1 \to 0 \to 0$; for the consumer-path fixture, they are $0 \to 1 \to 0$. The restored reference fixture's HTML is byte-for-byte identical to its pre-edit output, as verified through actual Sphinx builds.

SymPy: a new range constraint and existing product reduction. `sympy__sympy-12419` adds an optional range to `KroneckerDelta`, allowing sums over identity-matrix elements to use dimension bounds; the old `_remove_multiple_delta` and `_simplify_delta` still reconstruct two-argument objects, discarding the range. Let $d = \mathrm{KroneckerDelta}(i, j, (0, 2))$ and $P = \mathrm{Product}(d, (i, a, b))$, then substitute $a = b = j = 5$. With the original patch, reducing P before substitution yields 1, whereas substituting before evaluating the product yields the correct value 0. This input uses a finite-range delta accepted by the new interface, and its construction requires no out-of-bounds indexing into an identity matrix. After the two consumers conservatively retain expressions for range-constrained objects, both paths yield 0; controls for the old two-argument form, in-range equality/inequality, empty products, and the original task's identity-matrix summation are all preserved. The original starting revision supports only two arguments, so this case tests semantic preservation when a new representation enters existing consumers. The conservative repair aims to preserve range semantics and may reduce symbolic simplification; other local errors in the new API remain outside the scope of the repair. All 29 existing local delta and tensor-function tests pass at each of the three stages; the original model patch passes all $1 + 25$ tests in the official evaluation we executed. The execution environment uses Python 3.9.25 and mpmath 1.1.0.

xarray: the scope of consumption for variable and coordinate attributes. To repair scalar inputs, `pydata__xarray-6461` changes the attribute callback to capture the top-level variable's `x.attrs`. The existing coordinate-construction and merge paths reuse the same callback, thereby writing the variable's `units=K` into a time coordinate that originally has `units=s`, while losing the coordinate's own `axis=time`. The replay uses actual `xr.where` calls to construct arrays with variable and coordinate attributes; preserving and restoring the corresponding attributes by coordinate name restores the target coordinate's original values while preserving numerical results and the scalar-input repair. Controls with no attributes, variable attributes only, and coordinate attributes check numerical values and attributes separately. The execution environment uses Python 3.12, NumPy 1.26.4, and pandas 2.2.3, with a dependency configuration that differs from the original evaluation container.

Problem-statement correction. The original problem statement for `django__django-13513` was inconsistent with the actual test objective and has been corrected. Both the baseline reevaluation and the incremental run for this instance use the same corrected statement, and both pass evaluation; the correction and input-verification records are retained in the SWE-bench Verified study archive.

F.4 Reported Evidence and Scope of Inference

This appendix reports the archived cohort outcomes, the diagnoses for 38 originally failing runs, and the execution results and boundaries of the selected local replays. The analysis draws on the study archive's trajectories, patches, evaluation logs, and replay results; Appendix E lists the contents of the submitted review package. Sphinx builds use an isolated Python 3.12.3 environment with pinned dependencies.

These instances confirm specific evaluation blind spots in another benchmark and coding framework. The two datasets use the same model, while tasks, run budgets, and frameworks change together; the independent causal effect of architecture requires controlled experiments. Within the 500-run collection, our analysis covers human-reviewed evidence-based diagnoses of the 38 originally failing runs, including two terminal propagation omissions (one with a local repair replay), and execution-based validation of three purposively selected successful patches. The scope of inference is defined in Section 7. We executed both the official evaluation of the original model patches and the program replays before and after consumer synchronization, retaining inputs, patches, environments, and outputs separately. The original evaluation columns in Table 13 correspond to the original model patches, whereas the local synchronization results correspond to the target checks and controls listed for each case. The original 62 evaluation tests for Sphinx 9461 were also executed separately before and after consumer synchronization.

G Terminal Mismatches and Recovery on FeatureBench with Qwen3.8-Flash

G.1 Selection, Run Settings, and Evidence Method

This extension uses archived FeatureBench tasks (Zhou et al., 2026) run with `openai/qwen3.8-flash` and `mini-swe-agent 2.4.6`; the prefix is the framework's provider tag. Of the original 100 tasks, 46 completed an answer with status `Submitted`, and the remaining 54 exceeded the answer-generation time limit with status `TimeExceeded`. We analyze only these 46 completed first attempts across 15 repositories, fixed before retries; the 54 timed-out answers are excluded. `Submitted` records completion of the agent's answer, not passage of the task evaluation: the original evaluator marks 9 of these 46 runs resolved and 37 failed. This subset does not support a population mismatch-rate estimate or a model comparison.

We manually reviewed the task requirements, source shown in the archived trajectories, message-level tool calls and outputs, and final patches, and adjudicated the three terminal diagnoses and two additional recovery cases. The DeepSWE three-reviewer blinded procedure and its 97%/89% agreement figures do not apply to this extension. We manually confirmed terminal diagnoses in three runs: Meson dependency naming, MLflow host patterns, and Seaborn Path styling. The first two consumer relations have static evidence; the third has execution support from the archived original evaluation. Four separate runs contain five execution-supported recovery events: Seaborn plotting,

xarray tree nodes (two events), xarray chunk alignment, and scikit-learn nested validation. Human adjudication confirmed the latter two recovery cases while retaining the scope and classification boundaries described below. All executions are from the original trajectories or archived evaluations; we performed no new program executions or repair replays for these cases. The shared framework version and different model/benchmark support additional instances, without isolating model or framework effects.

All message references below are zero-based indices in `traj.json`; patch line numbers are physical lines in `final_patch.diff`.

G.2 Three Human-Confirmed Terminal Diagnoses

Table 14: Three FeatureBench terminal diagnoses. Evidence labels refer to the diagnosed existing consumer path. Original evaluation failures do not by themselves establish execution evidence for that path. All three runs fail the original evaluation.

Task	Final F2P; P2P	Retained consumer relation	Evidence
Meson Cargo	13/37; 103/103	New dependency names and wrap registrations disagree with existing resolver queries	Static
MLflow security	28/30; 94/94	New default IPv6 patterns misfeed the shared matcher used by existing FastAPI host validation	Static
Seaborn Path styles	28/29; 199/199	Shared capstyle adaptation discards the dashed-path theme setting	Execution

Meson: dependency registration and resolver lookup. The task is `mesonbuild__meson.f5d81d07.cargotests.8e49c2d0.lv1`. Δ is the new `_dependency_name` and Cargo-lock wrap generation; O is agreement between registered dependency names and the names queried by existing resolution paths; C is `Interpreter._fetch_package` $\rightarrow$ `_fetch_package_from_subproject` $\rightarrow$ `find_dep_provider`. D is the static mismatch between a normalized name such as `foo_1-rs` and the wrap's registered `foo-1-rs`. The git branch likewise registers package/API names while the existing `_dep_package` path queries repository-directory names. Messages 7, 39–40, 62, 107, and 139 locate normalization, callers, registration, and lookup; messages 141 and 143 display parser/wrap outputs. Final-patch lines 109–128 and 171–173 retain these naming decisions. We treat the related naming manifestations as one diagnosed run, without counting them as independent samples. The archived Cargo test failures are separate from this static consumer diagnosis; no execution trace confirms the complete resolver path here.

MLflow: default IPv6 patterns and existing host validation. The task is `mlflow__mlflow.93dab383.test_security.98fcde1d.lv1`. Δ is generation of a default `[::1]:*` pattern together with shell-style matching; O is to preserve acceptance of the intended localhost-with-port values when defaults reach existing host validation; C is `HostValidationMiddleware.__call__` in `mlflow/server/fastapi_security.py`; D is the mismatch between a literal IPv6 bracket and the bracket syntax consumed by `fnmatch`. Messages 7 and 9 retain the original localhost constants and FastAPI consumer. Final-patch lines 203–229 generate defaults and 232–260 define matching. The archived evaluation specifically expects the escaped default `[[]::1]:*` and observes the helper returning false for `[::1]:8080`. These executions confirm the default/helper defect; the existing middleware's rejection branch is established statically, not by an executed FastAPI HTTP 403 response.

Seaborn: shared adaptation loses an existing Path theme setting. The task is `mwaskom__seaborn.7001ebe7.test_line.25304a8b.lv1`. Δ is shared `Mark._handle_capstyle` adaptation; O is preservation of the separate dashed-line capstyle supplied by a plot theme; C is the existing `Path._plot` path constructing a Matplotlib `Line2D`; D is an executed dashed plot whose capstyle is `butt` instead of the requested `round`. Message 6 shows the original consumer; final-patch lines 138–152 retain the adapter. It falls

back to `lines.solid_capstyle='projecting'` and maps the dashed/projecting combination to `dash_capstyle='butt'`, losing `lines.dash_capstyle='round'`. The original study-archive file `tests/_marks/test_line.py`, lines 124–132, executes `Plot(...).add(Path(linestyle='--')).theme(rc).plot()`; `evaluation_logs/test_output.txt`, lines 14–20, records the resulting `Line2D` assertion failure. This execution witness concerns theme settings, not the later explicit artist-keyword override check that the failed test never reaches.

G.3 Four Runs with Recovery in the Original Trajectories

Table 15: Four archived FeatureBench recovery runs (five events). F2P/P2P are passing counts from the original final evaluation; all four runs remain unresolved. Recovery checks were executed by the original agent and are separate from the final evaluation.

Task	Final F2P; P2P	Consumer failure in the trajectory	Archived recovery evidence
Seaborn plotting	196/202; 245/245	Coordinate removal makes `Bar._make_patches` raise `KeyError: 'x'`	Same plotting script succeeds after two guards; 536 passed, 3 skipped, 1 xfailed
xarray tree nodes	33/34; 40/40	Detachment invalidates a live dictionary view, then rejects the existing delete-before-orphan path	Dictionary rebinding and removal of the strict key-count check; 153 passed, 5 xfailed
xarray chunks	14/18; 101/101	Early return leaves overlapping Zarr writes; readback loses the last row	Remove single-backend-chunk early return; 6 passed, 3 skipped
scikit-learn validation	215/218; 788/788	New wrapper's validation context rejects the existing internal LARS call	Skip nested validation as on the estimator path; 37 passed

G.4 Seaborn: Coordinate Handling Breaks an Existing Bar Consumer

The task identifier is `mwaskom__seaborn.7001ebe7.test_plot.b645d353.lv1`. Its statement requests restoration of declarative plotting interfaces, including paired variables and transformations. For $e = (\Delta, O, C, D)$, Δ is the edit to existing coordinate handling in `Plotter._compute_stats` and `_plot_layer`: an existing coordinate column is dropped before renaming a paired variable. O is to preserve coordinates already produced by a statistical transform when its output no longer contains the original paired-variable column. C is the existing `Bar._plot` → `Bar._make_patches` path, which reads `data[other]` to construct bars. D is the same plotting script's success, failure at this consumer after the edit, and success after guarded synchronization.

Initial state and failure. Message 137 creates synthetic data using `RandomState(0)`, generating `penguins` before `tips`; reproducing the inputs requires that generation order. Messages 145 and 149 construct and execute `/tmp/work/t2.py`, including `Plot(tips, y="total_bill").pair(x=["size", "day"]).add(Bar(), Hist())` followed by `plot()`. Message 150 reports `stat+pair ok 2`. This is an earlier working state within the trajectory, not an execution of the untouched FeatureBench starting repository. Message 155 applies the coordinate edit; message 156 reports `KeyError: 'x'`. Messages 159–160 rerun the same script and retain the traceback through `_plot_layer`, `Bar._plot`, and `Bar._make_patches`; the latter reads a coordinate column already removed upstream.

Recovery and final-patch binding. Message 163 guards both coordinate-removal sites by requiring the paired-variable column to exist as well: `var in df and axis in df`, with the corresponding condition on `out_df`. Message 164 records `stat+pair ok 2` from the same script and 536 passed, 3 skipped, and 1 xfailed in the agent's core/mark/stat tests. Final-patch lines 1059–1078 retain both guards (conditions at lines 1066 and 1076). This is one recovered consumer

relation, not a terminal omission. The original final evaluation still has six F2P failures (Table 15); recovery of this relation does not imply task success.

G.5 XARRAY: RESTORED DETACHMENT CONFLICTS WITH TWO EXISTING DELETION PATHS

The task identifier is `pydata__xarray.97f3a746.test_treenode.aa8ba777.lv1`. Its statement requests restoration of tree-node operations, including parent reassignment and orphaning. Messages 6 and 8 preserve the original consumers before the new detach implementation: `TreeNode.children.deleter` iterates over `self.children.values()` and calls `child.orphan()`, whereas `TreeNode.__delitem__` deletes a mapping key before calling `child.orphan()`. The `DataTree` delegation is also recorded in the trajectory. Message 146 creates the first implementation block for `_detach_child`; message 158 installs it in `xarray/core/treenode.py`. It deletes dictionary entries in place and requires exactly one key for the child. Here the consumers predate the edit, but there is no pre-edit successful execution witness for the restored operation.

Event 1: preserving an active consumer's iteration. Δ is in-place dictionary deletion by the new detach operation; O is to preserve the old mapping view while the existing bulk-deletion consumer iterates through it; C is `children.deleter`, reached through the setter and `orphan()`; D is the runtime failure of the existing `test_is_hollow` test. The original source at message 6 shows this consumer at lines 186–194. Messages 186–187 execute the DataTree tests and record `RuntimeError: dictionary changed size during iteration` at this path (1 failed, 17 passed). Message 188 changes detachment to rebind `self._children` to a filtered new dictionary. Message 189 advances beyond that test to the second consumer failure; message 191 subsequently confirms the whole test file passes apart from expected failures.

Event 2: respecting delete-before-orphan ordering. Δ is the same operation's strict one-key precondition; O is to accept the already-deleted-key state produced by an existing caller; C is `TreeNode.__delitem__`, shown at original source lines 649–656 in message 8; D is the existing `test_delitem` failure in message 189, reporting zero keys where one was required (1 failed, 55 passed, 4 xfailed). Message 190 removes the strict key-count check while retaining dictionary rebinding. Message 191 reports 153 passed and 5 xfailed for `xarray/tests/test_datatree.py`. Final-patch lines 232–238 retain both changes. The two obligations belong to distinct existing consumer paths within one file and count as two recovered events in one run, not two independent samples. The final evaluation still fails the duplicate-child F2P test: an `AssertionError` occurs where `InvalidTreeError` is required ($33/34$ F2P, $40/40$ P2P), separate from these recovered relations.

G.6 TWO ADDITIONAL HUMAN-CONFIRMED RECOVERY CASES

xarray: chunk alignment and an existing Zarr writer. The task is `pydata__xarray.97f3a746.test_backends_chunks.fa55f68a.lv1`. Δ is the early return in the restored `_align_nd_chunks`, introduced in message 44, that leaves variable chunks unchanged for a single backend chunk; O is to preserve stored values by keeping concurrent writes to a storage chunk from overlapping; C is the existing `Dataset.to_zarr` and Zarr backend path with `align_chunks=True`; D is executed data corruption followed by recovery after removal of that early return. Message 101 records six failed Zarr alignment tests and three skips. For a 2×2 array, message 109 observes readback `[[0,1],[0,0]]` instead of `[[0,1],[2,3]]`; message 113 provides a successful single-Dask-chunk control. Message 114 removes the single-backend-chunk early return, and message 117 reports the same group with six passed and three skipped. Final-patch lines 300–303 retain only the single-variable-chunk early return, preserving the recovery. We manually confirmed this recovered consumer relation while retaining a task-scope conflict: message 1 explicitly requires dimensions with one backend chunk to be returned unchanged. The recovery addresses the existing writer's data-integrity obligation but changes that stated special case; confirmation does not make the scope undisputed. The final evaluation retains four direct chunk-grid failures ($14/18$ F2P, $101/101$ P2P).

scikit-learn: a restored wrapper changes nested validation. The task is `scikit-learn__scikit-learn.5741bac9.test_public_functions.28421aef.lv1`. Δ is the restored `graphical_lasso` wrapper with `prefer_skip_nested_validation=False` in messages 181 and 183; O is to preserve the validation context required by the existing internal LARS call when exposing the algorithm through the public wrapper; C is `_graphical_lasso` $\rightarrow$ `lars_path_gram(method="lars")`, shown before the wrapper edit in message 103; D is an executed `InvalidParameterError` and recovery under the estimator's validation context. Message 186 records five failed and 32 passed covariance tests; message 188 shows nested validation rejecting `method='lars'`. Message 190 runs the successful `GraphicalLasso(mode='lars').fit` control. Message 191 changes only the wrapper's nested-validation setting to `True`; message 192 reports 37 passed. Final-patch lines 1279–1288 retain that setting, and lines 1524–1533 retain the direct internal call. We manually confirmed the recovery as an execution-context dependency on an existing consumer. The classification boundary is that the repair lies in the newly restored wrapper's own configuration; this is distinct from the explicit task-scope conflict in the xarray case. The original final evaluation still has three wrapper/estimator parameter-validation failures ($215/218$ F2P, $788/788$ P2P), separate from the recovered LARS path.

G.7 Evidence Scope

The case descriptions above provide the task identifiers, edit–obligation–consumer–diagnosis chains, message and patch coordinates, original evaluation outcomes, and static/execution distinctions for the three terminal diagnoses and four recovery runs. The coordinates identify the original trajectories and final patches in the FeatureBench study archive; evaluator logs and test sources explain the evidence for each case. Appendix E lists the contents of the submitted review package.