EXECUTION-GRAPH EVIDENCE FOR HARD FAULT LOCALISATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Coding agents increasingly execute sequences of dependent jobs, where an upstream error can propagate through intermediate artefacts and become visible only much later. Diagnosing such failures requires retrospective fault credit assignment: mapping an incorrect downstream result back to the earlier executed function that introduced it. We investigate whether captured execution evidence—executed functions, intermediate runtime states, and exact artefact hand-offs—helps LLM reviewers localise subtle, data-dependent faults, such as threshold-boundary errors, and snapshot-selection errors, whose effects depend on the values encountered during execution.

In a controlled four-job study, reviewers received complete source code and top-level inputs and outputs for the full workflow, with conditions additionally exposing different forms of execution evidence (from source code, inputs/outputs and logs through to execution graph and intermediate artefacts and data states). For the easier fault mechanisms, additional execution evidence produced no comparable improvement in localisation. For the harder faults, reviewers were allowed to inspect additional runtime evidence in stages: first selecting a relevant captured execution group, then opening a specific intermediate artefact. On the hard instance, this process led to the correct fault localisation in four sessions, whereas extra reasoning calls without any new runtime evidence finished at 0/6.

These results suggest a broader use for execution graphs beyond observability and governance. An agent could use the execution graph to trace a downstream failure back through its own execution and retrieve the specific runtime state needed to diagnose it. This points toward execution history being used not only to explain agent behaviour after the fact, but potentially to help the agent identify where its own workflow went wrong and what should be revisited.

1 MOTIVATION AND SCOPE

Complex agents, including long-horizon coding agents, work through dependent jobs and may pass results between sessions or collaborating instances. This makes early errors costly: a plausible but incorrect output can be reused by later jobs and remain unnoticed until near the end. By then, the accumulated context may make it difficult to identify the root cause or learn from the failure.

The broader question we ask is whether an agent can use a code execution graph of its own code trajectory to ground its memory of what actually happened during execution. By capturing intermediate data states and linking them to the functions that produced them, the graph leaves a runtime trail that the agent can follow back when a later result goes wrong. We test this broader idea through two questions.

Question 1: can execution graphs support fault localisation? The execution graph records the data states and intermediate values produced during execution, the functions that produced them, and the relationships between those states. The reviewer LLM is exposed to this graph through different representations and levels of detail. We test whether those representations help the reviewer identify exactly where in the code the upstream fault responsible for a downstream error was introduced.

Question 2: which types of error are more likely to need execution-graph evidence for localisation, and how much execution detail is useful? Some faults are already clear from source and top-level I/O, while others depend on runtime values or intermediate transformations that those views do not expose. We therefore compare compact execution-graph evidence with progressively revealed execution groups and intermediate artefacts.

We test these questions using controlled mutations of a marketing workflow. Each faulty instance introduces one subtle, data-dependent error - for example, premature contribution rounding or retaining the oldest repeated observation - and allows the resulting difference to propagate through the rest of the workflow. Every reviewer sees the complete source and top-level inputs and outputs, so the experiment asks whether execution evidence helps when the usual code and workflow-level evidence is already available.

The same execution record could also support human audit and governance by showing how data moved through the code, but we do not evaluate those uses here.

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.

2 Related Work

Structured agent systems retain state beyond the current transcript. MAGE [3] maintains a hierarchical execution-state tree, while AgentRewind [19] restores aligned agent and environment checkpoints. These systems primarily support continuity or recovery. More directly, Who&When [18] formulates failure attribution as identifying the responsible agent and decisive error step, and TraceElephant [4] studies attribution from fuller execution traces. Recent work on FALAT and Adaptive Influence Graphs follows trajectory dependencies or restructures failed traces to separate error introduction from propagation [1, 15]. These works operate mainly over agents and conversational steps; we target the executed Python function at which an error entered a multi-job workflow, using captured within-job relationships and exact-hash cross-job handoffs.

Classical program analysis treats localisation as backward reasoning over dependencies. Program dependence graphs represent data and control dependences [7], dynamic slicing restricts them to a particular execution [13], and Thin Slicing begins with value-producing statements before hierarchically expanding explanatory dependencies [16]. Workflow-provenance research makes a related distinction between fine-grained capture and semantic views: Dey et al. [5] map noWorkflow execution traces onto user-annotated YesWorkflow descriptions because statement-level provenance can overwhelm users.

Graph-to-LLM research likewise shows that representation matters. Talk Like a Graph finds that graph-task performance varies substantially with textual encoding [6]. Huang et al. [10] find that LLMs may treat natural-language graph descriptions as contextual text rather than recover the intended structure. Learned approaches such as GraphAdapter fuse graph and language representations [9], while G-Retriever selects a question-relevant connected subgraph [8]. We instead use a model-agnostic textual interface to an observed execution graph and compare compact evidence with progressively finer runtime disclosure.

Coding-agent work provides complementary localisation methods. LocAgent and RepoGraph use static repository graphs [2, 14], AutoFL lets an LLM navigate a repository through tool calls [12], and TraceCoder and DebugRepair use runtime evidence to diagnose or repair individual programs [11, 17].

3 Agent Architecture

The experiment uses a marketing pipeline. Each job executes in a fresh session and passes structured artefacts to the next job. The execution graph for each job is deterministically derived from the observed runtime execution rather than inferred by the reviewer: it records the functions that actually ran, the states they consumed and produced, and cross-job artefact handoffs only when upstream outputs exactly match downstream inputs by SHA-256 hash. Twelve faulty instances cover six executed mutation mechanisms across Jobs 1 and 3: premature contribution rounding, two snapshot-time errors, two repeat-resolution errors, and a non-cumulative channel cap. Each mutation preserves the schema and introduces a nonzero change that propagates to the final output. Review occurs in a separate session after execution.

All conditions are isolated from each other and from the working repository. Each review uses a fresh model session containing only the common task and one frozen condition-specific package. No answer, package, prompt, or session history is carried across conditions or repetitions; no repository path, filesystem tool, or undeclared source is available. Progressive follow-ups remain inside their originating review.

4 Execution Graph and Evidence Interface

4.1 What the execution graph records

Runtime capture gives each job a separate graph $G_j = (V_j, E_j)$. State nodes contain actual inputs, intermediate values, or outputs. FunctionMapping nodes identify the executed functions that produced or transformed those states. Relationships record observed flow between them, such as a function consuming one state and producing another. A multi-job chain is represented as

$$C = \left(\bigsqcup_j G_j, A \right),$$

where A contains typed controller handoffs created only from exact SHA-256 matches between upstream outputs and downstream inputs. These handoffs are separate from the captured within-job relationships and do not assert any other dependency between jobs.

4.2 REVIEWER-FACING EVIDENCE

Every condition receives byte-identical complete workflow source and actual top-level inputs and outputs across the workflow. The experimental conditions differ only in the additional execution evidence or interaction available to the reviewer.

The conditions vary three things on top of this common baseline. Compact execution evidence exposes executed-function anchors, bounded runtime values, execution status, handoff identifiers, and descriptors of nested computation. Progressive runtime inspection starts from that compact view and lets the reviewer first select one nested execution group and then inspect one intermediate artefact from that group. P03 Compact Graph + Graph Summary and P05 Progressive Graph Inspection + Graph Summary also receive a 14-item Graph Summary. Each item names an executed function and job, lists its expected input and output types, and gives a one-sentence description of its purpose. The Graph Summary describes intended roles but adds no runtime values or relationships. P06 Equal-Call Reconsideration adds reviewer calls without exposing new runtime evidence, allowing additional reasoning time to be compared with additional evidence.

Table 1: Reviewer conditions. Every condition includes complete source and all-job top-level I/O.

Condition	Additional evidence or interaction	Calls/session
P01 No Graph	No execution graph	1
P02 Compact Graph	Compact native execution graph	1
P03 Compact Graph + Graph Summary	P02 Compact Graph plus the Graph Summary	1
P04 Progressive Graph Inspection	P02 Compact Graph start, one selected nested-group expansion, and one selected artifact inspection	3
P05 Progressive Graph Inspection + Graph Summary	P04 Progressive Graph Inspection plus the Graph Summary	3
P06 Equal-Call Reconsideration	P02 Compact Graph start plus two no-new-evidence reconsiderations	3

P01 No Graph, P02 Compact Graph, and P03 Compact Graph + Graph Summary require one reviewer call, in which the reviewer receives its assigned evidence package and returns a terminal localisation. P04 Progressive Graph Inspection and P05 Progressive Graph Inspection + Graph Summary use three calls because the graph is navigated progressively rather than exposing all nested runtime state at once. The first call chooses a nested execution group, the second chooses an artefact from that group, and the third returns the terminal localisation after the selected artefact becomes visible. P06 Equal-Call Reconsideration also uses three calls, but reveals no new evidence: the reviewer gives an initial diagnosis and then reconsiders it twice using the same starting package. It therefore controls for additional reasoning opportunities without additional runtime state.

5 EXPERIMENTAL DESIGN AND REVIEW PROCEDURE

The study contains twelve one-site faults: two independently valued instances for each of six mechanisms, plus four independently valued clean controls. The controls run the same workflow without a mutation and are used to measure whether a reviewer incorrectly reports a fault when the output matches an independently generated clean reference. The faults are premature contribution rounding, recorded-time snapshot substitution, expiry compared with recorded time, retention of the oldest repeated observation, repeat ranking scoped only by source, and a non-cumulative Job-3 channel cap. Every condition receives byte-identical complete source and actual top-level inputs and parsed outputs for all four jobs; only Job 4 contributes stdout/stderr. The executed source contains no diagnostic-only witness tables: progressive inspection discloses naturally occurring states produced or consumed by the computation. The experiment therefore tests the incremental value of execution evidence over an unusually strong source-plus-all-job-I/O baseline.

In every condition, an LLM reviewer evaluates only its frozen package and returns one top suspect job and function. Exact localisation records whether that function matches the qualified mutation location. Reviewers do not receive the injected mutation, behavioural-oracle records, reference-generation records, or another condition's outputs.

Conditions are isolated during review. Each reviewer session receives only one frozen condition-specific package and cannot access the working repository, another condition's package or answer, or any previous reviewer session. The three repetitions for a package are also separate sessions. The only multi-call interactions are the follow-ups defined inside P04 Progressive Graph Inspection, P05 Progressive Graph Inspection + Graph

Summary, and P06 Equal-Call Reconsideration.

We freeze one package for each of the 96 instance–condition combinations and review every package in three fresh model sessions, producing 288 terminal judgements and 576 calls. In P04 Progressive Graph Inspection and P05 Progressive Graph Inspection + Graph Summary, the first two calls navigate to a graph group and then an artefact before the third call gives the terminal diagnosis. Any diagnosis text in those navigation steps is part of the interaction trace, not a separate result. P06 Equal-Call Reconsideration also uses three calls, but its follow-ups reveal no new evidence. The sessions are response-isolated stochastic draws from the reviewer model on the same package. Run-level comparisons use all 36 faulty reviews per condition while stratifying by the 12 fault packages.

The primary outcome is whether the reviewer names the exact upstream function in which the fault was injected. We also record whether it detects a fault, identifies the correct upstream job, raises a false alarm on a clean control, and—for P04 Progressive Graph Inspection and P05 Progressive Graph Inspection + Graph Summary—selects the designated graph group or artefact. Calls, tokens, latency, and cost measure resource use. The repeated sessions sample reviewer variability on fixed packages; they are separate model runs, but not newly generated faults or program executions.

We use *hard* as an empirical description, not as a label assigned from mutation design alone: a mechanism is hard here when substantial localisation difficulty remains under the complete-source, all-job-I/O P01 No Graph baseline. By that criterion, only premature contribution rounding is consistently hard in this experiment.

Before freeze, qualification checks verify that the mutations execute and propagate, the designated natural states are captured and reachable through the graph interface, complete source and all-job I/O are byte-identical across conditions, and fully rendered requests contain no forbidden answer fields. For each of the four controls, the outputs of the unmodified workflow exactly match a separately generated clean reference for the same inputs. This reference output—the clean oracle—is used only for qualification and is not shown to reviewers. Package-difference and replay audits preserve the frozen requests and responses.

6 RESULTS

Each condition has 36 reviews: three fresh LLM sessions for each of 12 frozen fault instances. We use all runs to estimate stochastic reviewer behaviour on these packages. Pairwise two-sided exact tests condition on total successes within each instance. Reported p-values are unadjusted.

6.1 EXECUTION-GRAPH EVIDENCE AND FAULT LOCALISATION

P05 Progressive Graph Inspection + Graph Summary produces the best overall localisation performance. Its advantage over P01 No Graph meets the conventional 5% significance threshold. P02 Compact Graph also performs better than P01 No Graph, but that smaller difference is not statistically significant. These comparisons provide encouraging evidence for Question 1: presenting the reviewer with captured execution-graph evidence can increase the likelihood of locating the function where a fault first appeared, even when complete source and workflow-level inputs and outputs are already available.

The improvement is concentrated in one fault mechanism, and P04 Progressive Graph Inspection does not consistently improve on P02 Compact Graph. Table 2 gives the full run-level comparisons.

Table 2: Run-level exact-localisation comparisons. Tests are two-sided, exact, stratified by fault package, and unadjusted. RD is the percentage-point risk difference.

Contrast	Left	Right	RD	Odds ratio	p
P05 Progressive Graph Inspection + Graph Summary vs P06 Equal-Call Reconsideration	34/36	28/36	+16.67	4.86	.0125
P05 Progressive Graph Inspection + Graph Summary vs P01 No Graph	34/36	30/36	+11.11	3.40	.0500
P05 Progressive Graph Inspection + Graph Summary vs P03 Compact Graph + Graph Summary	34/36	30/36	+11.11	3.40	.0500
P02 Compact Graph vs P01 No Graph	33/36	30/36	+8.33	2.20	.2000
P03 Compact Graph + Graph Summary vs P02 Compact Graph	30/36	33/36	−8.33	0.45	.2000
P04 Progressive Graph Inspection vs P02 Compact Graph	33/36	33/36	0.00	1.00	1.0000

6.2 THE GAINS OVER P01 NO GRAPH OCCUR ON ONE HARD MECHANISM

Premature contribution rounding is the only mechanism that remains consistently difficult under the strong source-and-I/O baseline. Every positive difference between P02 Compact Graph, P04 Progressive Graph In-

spection, or P05 Progressive Graph Inspection + Graph Summary and P01 No Graph occurs on the two rounding instances. The other five mechanisms are already at ceiling across the substantive conditions. In this experiment, then, the graph is useful precisely where source and top-level I/O leave residual ambiguity.

This also clarifies what we mean by a "hard" fault. The designation is based on post-review performance, not simply on an expectation that a mutation should be subtle. The rounding faults are empirically hard because reviewers often miss them without runtime evidence. The broader Question 2 remains open: with only two instances of this mechanism, the experiment cannot yet identify which general properties make a fault likely to benefit from execution-graph evidence.

In Table 3, Round is premature contribution rounding; Snapshot is recorded-time snapshot substitution; Expiry compares expiry with recorded time; Oldest retains the oldest repeated observation; Scoped ranks repeats only within source; and Cap is the non-cumulative Job-3 channel cap.

Table 3: Corrected exact localisation by mechanism. Each cell is exact reviews out of six; abbreviated mechanism headings are expanded in the text.

Condition	Round	Snapshot	Expiry	Oldest	Scoped	Cap
P01 No Graph	0/6	6/6	6/6	6/6	6/6	6/6
P02 Compact Graph	3/6	6/6	6/6	6/6	6/6	6/6
P03 Compact Graph + Graph Summary	0/6	6/6	6/6	6/6	6/6	6/6
P04 Progressive Graph Inspection	3/6	6/6	6/6	6/6	6/6	6/6
P05 Progressive Graph Inspection + Graph Summary	4/6	6/6	6/6	6/6	6/6	6/6
P06 Equal-Call Reconsideration	0/6	6/6	6/6	4/6	6/6	6/6

The four clean controls are reviewed three times in every condition. None of the 72 control reviews reports a fault, so the localisation gains are not accompanied by additional false alarms. The aggregate results use corrected scores: the original scoring script mistakenly expected Job 1 for two faults that were actually injected in Job 3 and therefore marked correct Job-3 answers as wrong. Re-scoring the frozen answers against the preserved fault records adds the same six successes to every condition, so none of the differences between conditions changes. Appendix B gives the audit details.

6.3 Additional calls do not explain the best result

P06 Equal-Call Reconsideration controls for a narrower confound: P05 Progressive Graph Inspection + Graph Summary might appear better simply because it uses more LLM calls. Both conditions use three calls and begin from P02 Compact Graph, but P06 Equal-Call Reconsideration reveals no new evidence. P05 Progressive Graph Inspection + Graph Summary performs significantly better at the conventional 5% level. This makes additional calls alone an unlikely explanation for its higher score. The two extra errors in P06 Equal-Call Reconsideration on otherwise easy oldest-repeat faults arise when reconsideration overturns an initially correct answer.

This control does not establish that progressive inspection itself is the source of the improvement. P04 Progressive Graph Inspection ties P02 Compact Graph when the Graph Summary is absent, and the comparisons involving the Graph Summary are mixed. P05 Progressive Graph Inspection + Graph Summary combines graph operations with that summary, so the experiment does not separate their contributions. Reviewers usually select the designated natural state on the eligible repeat-resolution cases, but those cases are already easy with P01 No Graph; selection therefore confirms that the state is reachable and used, not that selecting it causes success.

6.4 Interpretation

The results of this experiment, considered alongside previous experiments not included in the present paper, provide directional support for using operations on code execution graphs to improve agent performance, particularly when reviewing long pipeline trajectories. This evidence is consistent with our initial hypothesis.

For Question 1, the results suggest that giving the reviewer agent a representation of the code execution graph can increase the likelihood of localising where a fault first appeared. The pattern is visible even in the comparatively simple architecture studied here, where the jobs and overall pipeline are considerably shorter and more direct than a genuinely long-horizon code trajectory.

For Question 2—whether the benefit is greater for particular "hard" faults, and what makes a fault hard before or after review—the answer remains ambiguous. Further work is needed to characterise the faults for which execution graphs add information beyond source-code analysis and to distinguish them from faults that are already straightforward to trace from source and top-level I/O.

A related question concerns which evidence package best supports LLM review. P02 Compact Graph does not establish a reliable improvement, suggesting that the reviewer may struggle to interpret the representation consistently without guidance. P05 Progressive Graph Inspection + Graph Summary produces the best observed performance. Graph adapters may offer another way to help models use this structure and should be investigated directly.

7 LIMITATIONS

The evidence comes from one synthetic marketing workflow, one reviewer model and configuration, twelve fault instances, and four clean controls. The workflow contains four relatively short jobs and is substantially shorter and less interconnected than the trajectories produced by genuinely long-running agents. The faults are controlled, schema-preserving numerical mutations that propagate through this particular pipeline; they do not cover the wider range of failures that may arise in practice, including crashes, missing artefacts, ambiguous requirements, state corruption, tool failures, or interacting faults. Each of the six fault mechanisms has only two independently valued instances.

Each condition has 36 fresh fault reviews, comprising three reviewer runs for each of the twelve fault instances, together with twelve clean-control reviews. This provides an initial sample of LLM review behaviour, but more fresh reviews are needed to measure variability reliably across models and prompts. More independent faults, mechanisms, datasets, workflows, and substantially longer pipelines are also needed to establish how broadly the results generalise.

More work is needed to isolate the effect of graph topology, especially through additional methods such as a more developed graph adapter that allows the LLM to interact with the graph more easily.

The study evaluates detection and localisation, not autonomous repair, resumption, production reliability, or human review.

8 CONCLUSION

The experiment provides encouraging evidence that exposing an execution graph can improve fault localisation when complete source and workflow-level I/O are not enough. The best observed performance comes from P05 Progressive Graph Inspection + Graph Summary, whose advantage over P01 No Graph meets the conventional 5% significance threshold. P06 Equal-Call Reconsideration shows that this result is not explained simply by allowing more LLM calls.

The finding remains narrow. Every positive difference between a graph condition and P01 No Graph occurs on premature contribution rounding, the only empirically hard mechanism in this experiment. P02 Compact Graph shows a smaller, non-significant improvement, and P04 Progressive Graph Inspection does not consistently outperform it. The study therefore supports execution-graph operations as a promising direction for reviewing long pipeline trajectories, but further work must determine which faults benefit and how additional graph representations, guided operations combined with semantic graph descriptions contribute.

The broader motivation is not simply better post-hoc debugging. Complex agents, such as long-horizon agents, accumulate code, data, artefacts, and decisions across many dependent jobs. An error introduced early can therefore remain plausible, be reused repeatedly, and become visible only after substantial downstream work has already been built on top of it. At that point, an increasingly long conversational history is a poor substitute for knowing what actually executed. A deterministic execution graph provides a separate operational record derived from runtime artefacts: which functions ran, which states they consumed and produced, and how source functions map to the intermediate artefacts created during execution.

That record creates a different way for an agent to use observability. Rather than receiving the entire execution history as undifferentiated context, the agent can first orient itself using a higher-level view and retrieve deeper runtime state only where needed. Conversely, detail associated with parts of the execution that have already been validated could remain collapsed rather than repeatedly consuming context.

This suggests a possible role for execution graphs as part of the control loop of longer-running agents, not only as a record for human observability or governance. A future agent could use captured execution history to retrace an error to the earliest relevant divergence, inspect the state at that boundary, repair the affected step, and resume only the downstream work that depends on it, while preserving work already established as valid. The present study evaluates localisation and evidence navigation only. Whether execution-aware context selection improves

repair, reduces wasted downstream work or context use, and enables reliable resumption remains to be tested directly.

AI USE STATEMENT

Generative AI tools were used as the experimental systems that produced the fault-localisation judgments and prescribed follow-up judgments analysed in this paper. Generative AI assistance was also used in developing and reviewing experimental code, preparing analysis and artifact materials, and drafting or editing research text. The authors reviewed the AI-assisted code, records, analyses, claims, and prose and take responsibility for the final content, including text, claims, code, and artifacts produced with generative-AI assistance.

ETHICS STATEMENT

This study analyses machine-generated judgments over synthetic controlled software faults and does not intentionally contain human-subject or personal data. The anonymous release excludes credentials, raw authentication material, workstation state, and identifying metadata. Relevant risks are overgeneralising model-generated software judgments, using unreliable diagnoses without human verification, and the resource cost of 576 LLM reviewer calls.

REPRODUCIBILITY STATEMENT

The anonymous results archive `attempt-057-corrected-results.zip` contains all 96 instance-by-condition cells, all 288 corrected review rows, the full-sample run-level significance table, the deterministic rescoring manifest, natural-state selections, and per-call token and latency records. The appendices document the design, scoring correction, statistical method, and resource use. Full external-model reruns require service access and may vary over time.

REFERENCES

[1] Bakish et al. Adaptive influence graphs for failure attribution in multi-agent systems, 2026.

[2] Chen et al. LocAgent: Graph-guided LLM agents for code localization. In *ACL*, 2025. https://aclanthology.org/2025.acl-long.426/.

[3] Chen et al. Beyond semantic organization: Memory as execution state management for long-horizon agents, 2026. https://arxiv.org/abs/2606.06090.

[4] Chen et al. Seeing the whole elephant: A benchmark for failure attribution in LLM-based multi-agent systems. In *ACL*, 2026.

[5] Dey et al. Linking prospective and retrospective provenance in scripts. In *Theory and Practice of Provenance*, 2015.

[6] Fatemi et al. Talk like a graph: Encoding graphs for large language models. In *ICLR*, 2024.

[7] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM TOPLAS*, 1987.

[8] He et al. G-Retriever: Retrieval-augmented generation for textual graph understanding and question answering. In *NeurIPS*, 2024.

[9] Huang et al. Can GNN be good adapter for LLMs? In *The Web Conference*, 2024.

[10] Huang et al. Can LLMs effectively leverage graph structural information through prompts, and why? *TMLR*, 2024.

[11] Huang et al. TraceCoder: A trace-driven multi-agent framework for automated debugging of LLM-generated code, 2026. https://arxiv.org/abs/2602.06875.

[12] Kang et al. A quantitative and qualitative evaluation of LLM-based explainable fault localization. In *ACM FSE*, 2024.

[13] Bogdan Korel and Janusz Laski. Dynamic program slicing. *Information Processing Letters*, 1988.

[14] Ouyang et al. RepoGraph: Enhancing AI software engineering with repository-level code graph. In *ICLR*, 2025.

[15] Rafi et al. FALAT: Tracing failures in LLM agent trajectories via dependency-guided search, 2026.

[16] Sridharan et al. Thin slicing. In *PLDI*, 2007.

[17] Wu et al. DebugRepair: Enhancing LLM-based automated program repair via self-directed debugging, 2026. https://arxiv.org/abs/2604.19305.

[18] Zhang et al. Which agent causes task failures and when? On automated failure attribution of LLM multi-agent systems. In *ICML*, 2025.

[19] Zhuang et al. AgentRewind: Recoverable execution for long-horizon LLM agents, 2026. https://arxiv.org/abs/2608.14380.

A Formal Graph Construction

This is the same execution graph summarised in Section 4, not a separate graph used only in the supplement. We retain the formal detail here to make the capture and matching rules reproducible without interrupting the main argument. For job j, the captured graph is $G_j = (V_j, E_j)$. Nodes retain canonical identifiers, node type, bounded captured state, execution status, source identity, and runtime-stack information. Edges are relationships observed within that job. The chain $C = (\bigsqcup_j G_j, A)$ preserves the separate within-job graphs and records exact controller handoffs in A.

Before execution, a declared function has

$$D_f = (b_f, \iota_f, r_f, X_f, Y_f), \qquad \iota_f = (j_f, s_f, q_f, \sigma_f),$$

where b_f is an opaque boundary identifier; j_f, s_f, q_f, and σ_f are the job, normalised source path, qualified function name, and normalised function-source SHA-256; and r_f, X_f, and Y_f describe role and expected interface. The controller accepts a match only when this static identity resolves to the observed function. Similar names, partial identities, ambiguous matches, and wrong-job matches are rejected before review.

Let P_f be the nonempty set of canonical runtime-stack prefixes admitted by that identity match, with $p \preceq \mathrm{stack}(v)$ denoting a prefix. The realised captured boundary contains the runtime nodes reached under those prefixes, the incident observed edges, its captured inputs and outputs, and nested execution prefixes. The controller does not construct missing runtime nodes or relationships from source or declarations.

B Full Experimental Details

B.1 Results files and audit trail

The anonymous archive places the complete results first so that every summary in the paper can be checked directly. Its principal files are:

- `complete-result-cells.csv`: every frozen instance–condition cell;
- `all-review-results-corrected.csv`: every individual review;
- `run-level-stratified-significance.csv`: the full-sample statistical comparisons; and
- `corrected-rescore.json`: the deterministic Job-3 reporting correction.

The archive also includes natural-state selections and per-call token and latency records.

Archive SHA-256:

`a759f9a344feba9938eec4fffe846b1d`

`3c8ea0d4ae2cac839427cc558f8a4f65`.

B.2 Design and condition names

The experiment uses six mutation mechanisms, with two independently valued fault instances per mechanism, plus four independently valued clean controls. All conditions receive the same complete source and top-level inputs and outputs. Table 4 gives the condition names used throughout the paper.

Table 4: Experimental conditions.

Condition	Additional evidence or interaction	Calls/review
P01 No Graph	No execution graph	1
P02 Compact Graph	Compact native execution graph	1
P03 Compact Graph + Graph Summary	P02 Compact Graph plus the Graph Summary	1
P04 Progressive Graph Inspection	P02 Compact Graph start, one selected group expansion, and one selected artefact	3
P05 Progressive Graph Inspection + Graph Summary	P04 Progressive Graph Inspection plus the Graph Summary	3
P06 Equal-Call Reconsideration	P02 Compact Graph start plus two reconsiderations with no new evidence	3

The executed source contains no purpose-built diagnostic tables. P04 Progressive Graph Inspection and P05 Progressive Graph Inspection + Graph Summary reveal ordinary runtime objects produced or consumed by the workflow. P02 Compact Graph and all three-call conditions begin from byte-identical graph bytes. Every frozen package is reviewed in three fresh, response-isolated model sessions. A session can access only its assigned frozen package, not the working repository, another condition, or a previous session.

B.3 Experiment scale and aggregate outcomes

Tables 5 and 6 collect the experiment counts and condition-level outcomes. Detection, upstream-job attribution, and exact-function localisation coincide in every condition. P01 No Graph, P02 Compact Graph, P03 Compact Graph + Graph Summary, P04 Progressive Graph Inspection, P05 Progressive Graph Inspection + Graph Summary, and P06 Equal-Call Reconsideration all produce zero false positives on the clean controls.

Table 5: Experiment inventory.

Item	Count
Fault mechanisms	6
Fault instances	12
Clean controls	4
Fresh four-job workflow executions	16
Fresh job captures	64
Frozen review packages	96
Reviewer sessions	288
LLM reviewer calls	576
Repairs	0

Table 6: Aggregate outcomes. Fault outcomes are out of 36 reviews; control false positives are out of 12.

Condition	Detect	Upstream job	Exact function	Control FP	Calls
P01 No Graph	30/36	30/36	30/36	0/12	48
P02 Compact Graph	33/36	33/36	33/36	0/12	48
P03 Compact Graph + Graph Summary	30/36	30/36	30/36	0/12	48
P04 Progressive Graph Inspection	33/36	33/36	33/36	0/12	144
P05 Progressive Graph Inspection + Graph Summary	34/36	34/36	34/36	0/12	144
P06 Equal-Call Reconsideration	28/36	28/36	28/36	0/12	144

B.4 Fault mechanisms and selected runtime state

Table 7 shows that premature contribution rounding is the only mechanism with substantial residual difficulty. Every positive difference between a graph condition and P01 No Graph appears on its two instances. The other mechanisms are at ceiling in all substantive conditions; P06 Equal-Call Reconsideration alone loses two initially correct oldest-repeat answers during reconsideration.

In Table 7, Round is premature contribution rounding; Snapshot is recorded-time snapshot substitution; Expiry compares expiry with recorded time; Oldest retains the oldest repeated observation; Scoped ranks repeats only within source; and Cap is the non-cumulative Job-3 channel cap.

Table 7: Exact localisation by fault mechanism. Each entry is out of six reviews.

Condition	Round	Snapshot	Expiry	Oldest	Scoped	Cap
P01 No Graph	0	6	6	6	6	6
P02 Compact Graph	3	6	6	6	6	6
P03 Compact Graph + Graph Summary	0	6	6	6	6	6
P04 Progressive Graph Inspection	3	6	6	6	6	6
P05 Progressive Graph Inspection + Graph Summary	4	6	6	6	6	6
P06 Equal-Call Reconsideration	0	6	6	4	6	6

For the two repeat-resolution mechanisms, P04 Progressive Graph Inspection and P05 Progressive Graph Inspection + Graph Summary usually select the designated natural state. Exact localisation is already perfect whether that state is selected or not, and P01 No Graph is also perfect on these mechanisms. The selection result therefore shows that the state is reachable and used, not that it causes success.

Table 8: Natural-state selection for the eligible repeat-resolution reviews.

Review group	Reviews	Exact localisations
Designated natural state selected	19	19
Another artefact selected	5	5
P01 No Graph reference	12	12

B.5 Full-sample statistical analysis

The paper uses one significance calculation. For each contrast, the exact two-sided test uses all 36 faulty reviews per condition and stratifies by the twelve frozen fault packages. Within each package, it conditions on the combined number of successful reviews and evaluates how those successes could be allocated between the two conditions. Table 9 reports the unadjusted results.

Table 9: Full-sample exact-localisation contrasts. The exact two-sided test uses all reviewer runs and stratifies by fault package.

Contrast	Left	Right	Difference	Odds ratio	p
P05 Progressive Graph Inspection + Graph Summary vs P01 No Graph	34/36	30/36	+11.11 pp	3.40	.0500
P02 Compact Graph vs P01 No Graph	33/36	30/36	+8.33 pp	2.20	.2000
P03 Compact Graph + Graph Summary vs P02 Compact Graph	30/36	33/36	−8.33 pp	0.45	.2000
P04 Progressive Graph Inspection vs P02 Compact Graph	33/36	33/36	0.00 pp	1.00	1.0000
P05 Progressive Graph Inspection + Graph Summary vs P03 Compact Graph + Graph Summary	34/36	30/36	+11.11 pp	3.40	.0500
P05 Progressive Graph Inspection + Graph Summary vs P06 Equal-Call Reconsideration	34/36	28/36	+16.67 pp	4.86	.0125

B.6 Qualification, reporting correction, and resource use

Qualification checks that every mutation executes and reaches the final output, the relevant runtime states are captured and reachable, packages are isolated, and rendered reviewer requests contain no answer fields. Exact replay verifies the frozen requests, responses, and provider receipts. There are no calibration calls or repair attempts.

A reporting-script error assigned the wrong expected job to two Job-3 fault instances. Deterministic rescoring of the preserved responses corrected 36 job and exact-function outcomes. Each condition gained six successes, so all between-condition differences and statistical comparisons were unchanged. The anonymous artifact contains both the correction record and corrected results.

Table 10: Reviewer-model resource use.

Resource	Total
Input tokens	33,711,418
Cached input tokens	6,051,840
Output tokens	588,572
LLM reviewer calls	576
Provider latency (seconds)	13,203.730