Code Agents Don't Understand Root Cause:Evaluating Code Agents on Linux Kernel Fault Localization

Anonymous ACL submission

Abstract

Code agents are increasingly used in debugging workflows, where fault localization (FL) is often evaluated as endpoint retrieval: whether a model names the faulty code location. This view is inadequate for Linux kernel debugging, where failure logs may expose downstream symptoms while the erroneous state is introduced earlier and propagates through various dependences. We introduce KernelPath, a 200-instance Linux 5.10 function-level FL benchmark that pairs each root cause function with a source-grounded state propagation path and a dependence layer. KernelPath evaluates both endpoint localization and ordered diagnostic process recovery. Experiments with SOTA agents show that agents often retrieve plausible candidates but remain unreliable at recovering the diagnostic process, with performance degrading sharply under complex dependence relations. Further diagnostic probes show that agents can use explicit state-transition evidence, yet fail to infer how functions affect the erroneous state, drifting toward neighboring objectives such as repair-site selection, or upstream suspicious function retrieval. These findings reveal an unstable task model for kernel FL in current code agents.

1 Introduction

Code agents are increasingly used in software debugging workflows, where fault localization (FL) connects an observed failure to the code responsible for it and provides the basis for later repair (Wong et al., 2016; Zou et al., 2021). Early LLM-based debugging-agent evaluations mainly focused on repository-level software tasks (Jimenez et al., 2024; Yang et al., 2024b; Zhang et al., 2024; Xia et al., 2025; Wang et al., 2025). A growing trend is to move this evaluation toward system-level debugging, especially Linux kernel fault localization and crash resolution (Mathai et al., 2024; Zhou et al., 2025). The Linux kernel makes FL substantially harder: logs are sparse, symptoms often appear far from the root cause, and the failure-relevant state may propagate through complex mechanisms (Zeller, 2002; Cleve and Zeller, 2005; Ferrante et al., 1987; Horwitz et al., 1990; Zhou et al., 2025). These properties previously made kernel FL difficult to evaluate as an LLM task, but modern code agents can now search large repositories and inspect source context, making it possible to ask whether they can localize kernel root causes (Yang et al., 2024b; Wang et al., 2025; Zhou et al., 2025).

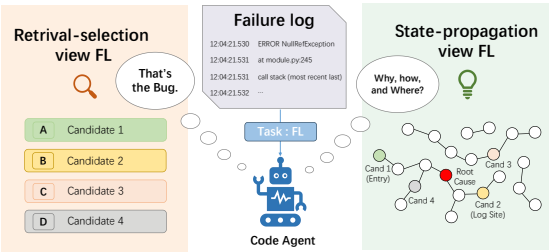

Figure 1: Two views of Linux kernel fault localization from the same failure log. The retrieval-selection view reduces FL to ranking candidate functions, whereas the state propagation view asks whether the agent can reconstruct how the state propagates.

Recent work has begun to follow this trend, but it still tends to focus on surface aspects of debugging. One line of work studies whether agents can retrieve or rank the correct fault location among many plausible candidates (Kang et al., 2024; Yang et al., 2024a; Rafi et al., 2024; Qin et al., 2025; Zhou et al., 2025); another studies whether agents can generate repairs or resolve debugging tasks through patch-oriented workflows (Jimenez et al., 2024; Yang et al., 2024b; Zhang et al., 2024; Xia et al., 2025;

Mathai et al., 2024). These evaluations show that current agents remain unreliable (Tian et al., 2024; Jimenez et al., 2024; Xia et al., 2025; Zhou et al., 2025), but their signals are still mostly endpoint- or repair-oriented. As a result, they explain failures as ranking errors, retrieval failures, or incomplete patch generation, rather than asking the more basic question: does the agent understand why a function is the root cause?

This creates a mismatch with Linux kernel FL. As shown in Figure 1, the retrieval-selection view treats FL as choosing one function from a ranked list of relevant candidates. However, kernel FL requires a state-propagation view, which is consistent with classical views of debugging as tracing cause-effect chains and dependence relations in programs (Zeller, 2002; Cleve and Zeller, 2005; Weiser, 1981; Ferrante et al., 1987; Horwitz et al., 1990). The log site may only expose the erroneous state; a downstream function may be a good place to block the symptom; and an upstream function may be necessary for reaching the failure path. All of them can look relevant, but they play different diagnostic roles. The root cause is the function where the failure-relevant state is introduced, committed, or first made wrong. Traditional endpoint metrics such as Top-k and MRR, which are widely used in prior FL evaluations (Kang et al., 2024; Yang et al., 2024a; Rafi et al., 2024; Qin et al., 2025; Zhou et al., 2025), cannot distinguish these roles, because they only measure whether the final answer hits an annotated function, not whether the agent recovers the diagnostic process that makes it the root cause.

To match this kernel-specific view of FL, we construct data around source-grounded state propagation rather than isolated endpoints. Each instance should specify not only the root-cause function, but also the source-grounded path through which the erroneous state propagates to the final log exposure. To measure this ability, endpoint accuracy is insufficient, so we introduce **Order-Weighted Path Recall (OWPR)** to evaluate whether the predicted diagnostic chain recovers the gold path in the correct order. Based on this **source-grounded state-propagation evaluation mechanism**, we construct **KernelPath**, the first **200-instance Linux kernel function-level FL benchmark** that jointly evaluates root-cause localization and diagnostic-process recovery.

Using KernelPath, we find that current code agents do not reliably understand root cause in Linux kernel FL. They can often retrieve relevant code, but fail to organize it into the state propagation process required by the task. This failure becomes more severe when propagation depends on complex relations such as cross-context or asynchronous mechanisms. Further diagnostic probes show the reason: agents struggle to assign the correct state-transition role to each function, and when this role attribution is unstable, they substitute neighboring objectives for FL. Repair evidence pulls them toward symptom-blocking locations, while explicit upstream prompts often move them toward upstream suspicious functions rather than the true root cause. Thus, their weakness is not merely poor retrieval, but an unstable task model of kernel root-cause localization.

The contributions of this paper are as follows:

- **We are the first to decouple root-cause endpoint localization from root-cause process understanding in code-agent FL evaluation.** This reframes kernel FL as identifying not only where the fault is, but why that function is the root cause.
- **We propose KernelPath, the first Linux kernel function-level FL benchmark for evaluating source-grounded root-cause understanding.** KernelPath implements a **source-grounded state-propagation evaluation mechanism** and introduces **OWPR** to measure ordered diagnostic-process recovery.
- **We reveal the benchmark conclusion that current code agents do not reliably understand root cause in Linux kernel FL.** They often retrieve relevant code, but confuse root-cause localization with repair-site selection or upstream suspicious-function retrieval.

2 Related Work

Fault localization from a state propagation perspective. Fault localization has

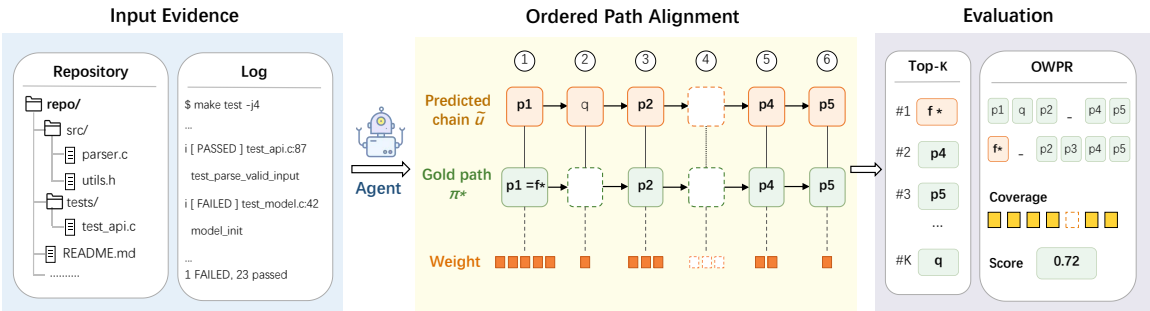

Figure 2: Evaluation protocol for endpoint localization and ordered diagnostic-process recovery. The agent receives the repository and failure log, returns ranked root-cause candidates and diagnostic chains, and is evaluated by Top-k endpoint localization and OWPR path alignment.

long been studied as a failure process in which an erroneous state is introduced, propagated, and eventually exposed (Zeller, 2002; Cleve and Zeller, 2005; Wong et al., 2016; Jones et al., 2002; Jones and Harrold, 2005; Liblit et al., 2005; Zou et al., 2021). Program slicing, dynamic slicing, program dependence graphs, and system dependence graphs further provide structured foundations for connecting program entities through data and control dependences (Weiser, 1981; Tip, 1995; Agrawal and Horgan, 1990; Agrawal et al., 1993; Ferrante et al., 1987; Horwitz et al., 1990; Zhang and Gupta, 2004). This literature motivates our view of kernel FL as source-grounded state propagation reconstruction, but our goal is not to propose a new slicing or dependence-analysis algorithm; instead, we evaluate whether modern code agents can use this view when localizing faults in the Linux kernel.

Code agents and kernel fault-localization evaluation. Code-agent evaluation has expanded from code generation and completion to repository-level software-engineering and debugging tasks (Chen et al., 2021; Lu et al., 2021; Liu et al., 2024; Guo et al., 2025; Jimenez et al., 2024; Yang et al., 2024b; Wang et al., 2025; Zhang et al., 2024; Xia et al., 2025; Tian et al., 2024; Kang et al., 2024; Yang et al., 2024a; Qin et al., 2025; Rafi et al., 2024; Zhou et al., 2025; Mathai et al., 2024). Closest to our setting, LinuxFLBench shows that agents still struggle with fine-grained localization in large and low-observability kernel codebases (Zhou et al., 2025). However, existing evaluations mainly rely on endpoint or repair-oriented signals such as Top-k, MRR, or repair success, which provide limited evidence about whether an agent recovers the diagnostic process supporting its FL judgment. KernelPath complements this line of work by pairing each root-cause function with a source-grounded diagnostic path and a dependence layer, enabling joint evaluation of endpoint localization and ordered state propagation recovery.

Prior work on LLM metacognition studies calibration, self-reflection, and self-correction (Flavell, 1979; Nelson, 1990; Kadavath et al., 2022; Shinn et al., 2023; Madaan et al., 2023; Huang et al., 2024; Elenjical et al., 2026; Ackerman, 2026; Servajean and Servajean, 2026; Shakarian, 2026). Here, we use task meta-knowledge more narrowly to ask whether agents consistently represent the kernel FL task, or instead drift toward adjacent debugging objectives despite retrieving relevant code.

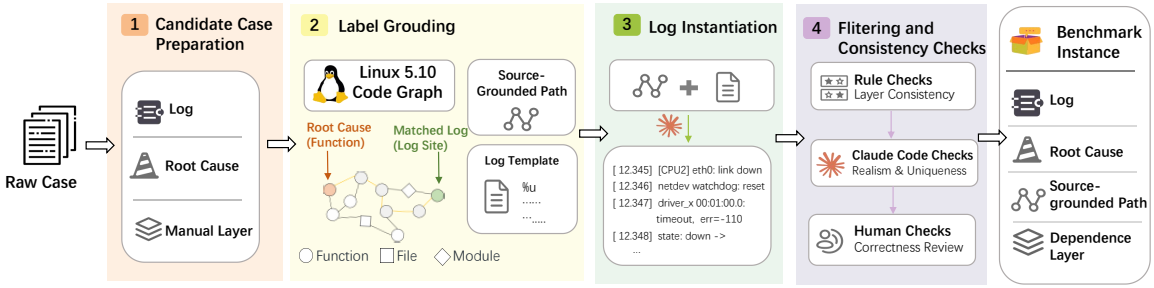

Figure 3: KernelPath construction pipeline. Candidate cases are normalized from raw kernel materials, grounded to Linux 5.10 Code Graph entities, paired with source-grounded diagnostic paths, instantiated as logs only after labels are fixed, and filtered through rule, Claude Code-assisted, and human consistency checks.

3 KernelPath

3.1 Task Formulation

Given a failure log D and a code repository R, the goal is to identify the root-cause function and reconstruct the diagnostic chain from the root cause to the log site. Formally, the agent outputs

$$(\hat{f}, u), \quad \hat{f} \in \mathcal{F}(R), \quad u = \langle u_1, \ldots, u_t \rangle,$$

where $\mathcal{F}(R)$ is the set of localizable functions in R, $\hat{f}$ is the predicted root-cause function, and u is the predicted function-level diagnostic chain.

3.2 Benchmark Framework

This section defines the framework of KernelPath. Following the task formulation in Section 4.1, each instance contains the failure evidence visible to the agent and hidden annotations used for evaluation and analysis. The framework specifies three components: the instance schema, the source-code grounding mechanism, and the dependence-based difficulty layer.

3.2.1 Benchmark Instance

Each instance consists of a failure log D, the Linux 5.10 repository R, and three hidden annotations: the root-cause function f^*, the source-grounded diagnostic path π^*, and the dependence layer $\ell \in \{L1, L2, L3\}$. Formally, an instance is represented as:

$$x = (D, R; f^*, \pi^*, \ell).$$

The agent observes only D and R. The label f^* is used to evaluate endpoint fault localization, while π^* records the function-level state propagation process from the root cause to the function containing the failure log. The layer ℓ supports layer-wise analysis of how dependence complexity affects agent performance.

3.2.2 Source-code Grounding

To make labels source-grounded, we build a Code Graph from Linux 5.10. The graph contains source-code entities such as functions, variables, fields, and log entities, together with structural relations such as calls, returns, assignments, field accesses, and log-message relations (Ferrante et al., 1987; Horwitz et al., 1990).

During annotation, the candidate log is first grounded to a log entity and then to the function that contains it. The candidate root cause is grounded to a function entity. Based on these endpoints, the graph supports path search from the root-cause function to the log site.

3.2.3 Dependence-Based Layers

The dependence layer characterizes the diagnostic difficulty of an instance. It is assigned according to the weakest dependence model required to certify the source-grounded diagnostic path π^*.

L1 covers local data/control dependence, where the failure-relevant state can be introduced, transformed, or checked within a local code scope. L2 covers same-context interprocedural propagation, where the state crosses function boundaries through calls, arguments, return values, fields, or same-context shared state. L3 covers cross-context or asynchronous propagation, where the state becomes failure-relevant through mechanisms such as IRQs,

timers, etc. This layer assignment depends on the dependence relations required by π^*, rather than on path length, log length, or model performance.

3.3 Construction

Figure 3 summarizes the construction pipeline of KernelPath. The main design principle is that labels are fixed before the log is instantiated. We first normalize Linux-kernel-related materials into candidate cases, each containing a candidate log site, a candidate root-cause function, and an initial dependence pattern. We then ground the candidate log site and root-cause function to the Linux 5.10 Code Graph, and search source-grounded paths between them. Based on the grounded endpoints and graph relations, we finalize the root-cause function f^*, select the diagnostic path π^*, and derive the dependence layer ℓ from the relations required by that path.

Only after f^*, π^*, and ℓ are fixed do we instantiate the log. Claude Code is used only in this post-certification stage to render Linux-style logs from repository and log templates; it does not determine the root cause function, diagnostic path, or dependence layer. Finally, we filter instances through rule checks, Claude-assisted inspection, and human review to remove cases with layer inconsistency, unrealistic logs, or non-unique root causes. Detailed audit rules and rejection standards are provided in Appendix A.3.We summarize KernelPath's positioning relative to representative fault localization and code repair benchmarks in Table 1; detailed release-split statistics are provided in Appendix A.6.

3.4 Evaluation Protocol and Metrics

Figure 2 summarizes our evaluation protocol. For each instance $x = (D, R; f^*, \pi^*, \ell)$, the agent observes only D and R. We parse its output into a ranked root-cause list $\hat{\mathbf{f}}$ and a predicted diagnostic chain u. Endpoint localization is measured by Top-k, which checks whether f^* appears among the top-k candidates in $\hat{\mathbf{f}}$.

Process recovery is measured by Order-Weighted Path Recall (OWPR), which aligns the predicted chain u with the gold source-grounded path $\pi^* = \langle p_1, \ldots, p_m \rangle$:

$$\mathrm{OWPR}(u, \pi^*) = \frac{\sum_{i=1}^{m} w_i c_i}{\sum_{i=1}^{m} w_i}. \tag{1}$$

Here, w_i is the decayed weight of the i-th gold node p_i, and c_i indicates whether p_i is recovered in the correct relative order:

$$c_i = \mathbf{1}[p_i \in u \wedge \rho_i > M_i]\,, \tag{2}$$

where

$$\rho_i = \mathrm{pos}_u(p_i), \quad M_i = \max_{\substack{j<i \\ p_j \in u}} \mathrm{pos}_u(p_j), \tag{3}$$

with $\max \emptyset = 0$. Thus, OWPR rewards a gold-path node only when it appears in the predicted chain and preserves the order of previously recovered gold nodes. We report OWPR@1 and OWPR@5, together with layer-wise results for L1–L3.

4 Experiments

We evaluate whether current code agents can perform Linux kernel FL by recovering the underlying state propagation process. The experiments answer three questions:

- **RQ1:** Can agents reliably recover the kernel failure-propagation process?
- **RQ2:** Where does an agent's FL decision fail?
- **RQ3:** When state propagation reconstruction is unstable, what is the agent actually optimizing?

4.1 Experimental Setup

Models. We evaluate four strong code-agent configurations: GPT-5.5/Codex, GPT-5.4/Codex, Qwen3.5-Plus/Qwen Coder, and Claude-4.7-Opus/Claude Code.

Protocol. Following Section 3.4, all agents receive the same failure log and Linux 5.10 repository access. RQ1 uses all 200 KernelPath instances. RQ2 and RQ3 use diagnostic subsets to isolate mechanisms; unless otherwise stated, subset probes use GPT-5.4/Codex, with full-agent results reported in Appendix A.8.

Benchmark	Task Setting		Evaluation Signal	
	Primary Task	Input	Process Supervision	Taxonomy
SWE-bench (Jimenez et al., 2024)	Repository resolution	issue + repo	–	–
DebugBench (Tian et al., 2024)	Code completion	buggy code + tests	–	–
LinuxFLBench (Zhou et al., 2025)	Linux kernel FL	bug reports + repo	–	–
KernelPath	Linux kernel FL	log + repo	OWPR	dependence

Table 1: Benchmark positioning. The comparison emphasizes evaluation signal and annotation structure rather than raw dataset scale. Here, repo denotes repository.

Model	Top-1	Top-5	OWPR@1	OWPR@5
GPT-5.5	46.5%	73.0%	61.72%	74.61%
GPT-5.4	47.0%	76.5%	53.84%	67.98%
Qwen3.5-Plus	49.0%	77.0%	59.59%	69.33%
Claude-4.7-Opus	53.5%	80.5%	68.95%	74.80%

Table 2: Overall FL and diagnostic-process recovery performance. OWPR@1 and OWPR@5 denote OWPR(top-1) and OWPR(top-5), respectively.

4.2 RQ1: Can Agents Reliably Recover the Kernel State Propagation Process?

4.2.1 Endpoint Accuracy Masks Process Failure

Endpoint localization overstates diagnostic ability. Table 2 shows that the strongest agent, Claude-4.7-Opus/Claude Code, reaches 53.5% Top-1 and 80.5% Top-5, but only 68.95% OWPR@1. The gap is larger for GPT-5.4/Codex: 76.5% Top-5 but 53.84% OWPR@1. Thus, agents often retrieve plausible functions without reconstructing the ordered state-propagation process that justifies the FL decision.

The process gap is distributional, not caused by a few failed cases. Figure 4(a) shows wide OWPR spread over the 200 instances even when endpoint retrieval is nontrivial. This indicates that process recovery is an independent evaluation signal rather than a stricter restatement of Top-k.

4.2.2 Dependence Complexity Breaks Process Recovery

Complex dependence is the main break point. Figure 4(b) shows a consistent L1-to-L3 drop for every agent. GPT-5.4/Codex falls from 75.40% OWPR@1 on L1 to 41.67% on L3, while Claude-4.7-Opus/Claude Code, the strongest configuration, still reaches only 56.76% on L3. The degradation therefore follows dependence complexity rather than model family.

The layer effect is not a path-length artifact. In our data, L3 chains are shorter on average than L2 chains, but Figure 4(b) still shows worse L3 recovery. Appendix Figure 9 gives the layer-by-depth distribution. The failure is therefore better explained by cross-context or asynchronous dependence, not by longer diagnostic chains.

Local edge recovery localizes the dependence bottleneck. Figure 4(b) shows that agents recover most L1 state-propagation edges, while L2 and L3 edge recovery drops sharply across all four agents. The process failure is therefore visible both in whole-path OWPR and in local edge recovery.

4.3 RQ2: Where Does an Agent's FL Decision Fail?

4.3.1 State Facts Provide the Missing Signal

State-transition facts, not paths alone, make FL decisions recoverable. We compare five inputs on a 30-case diagnostic subset: log only, path only, a neutral length-matched control, self-inferred state, and explicit state-transition facts. For GPT-5.4/Codex, accuracy stays near 20–23.3% under log-only, path-only, neutral, and self-state inputs, but rises to 90.0% when concrete state-transition facts are provided.

The state-aware shift is consistent across agents. Figure 6(a) shows that the state-aware condition produces the largest accu-

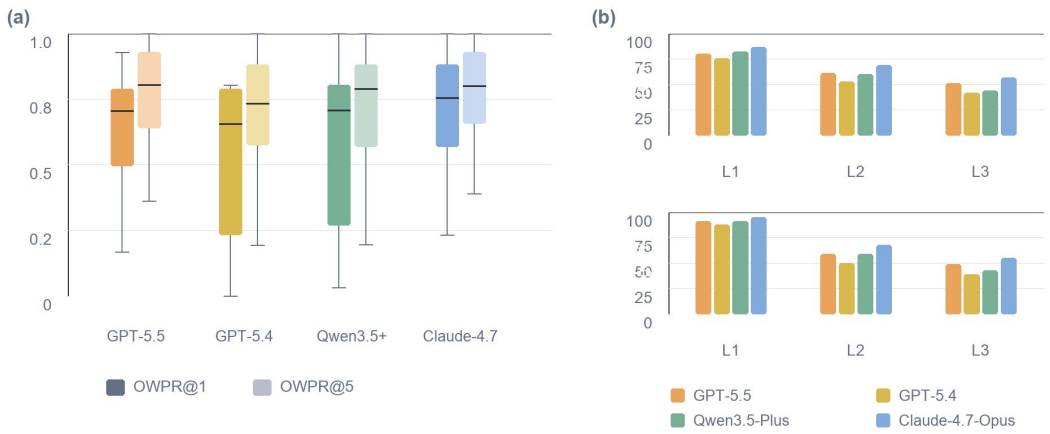

Figure 4: Process-recovery evidence for RQ1. Panel (a) shows full-instance OWPR distributions. Panel (b) shows that both layer-wise OWPR and state-propagation edge recovery decline from L1 to L3.

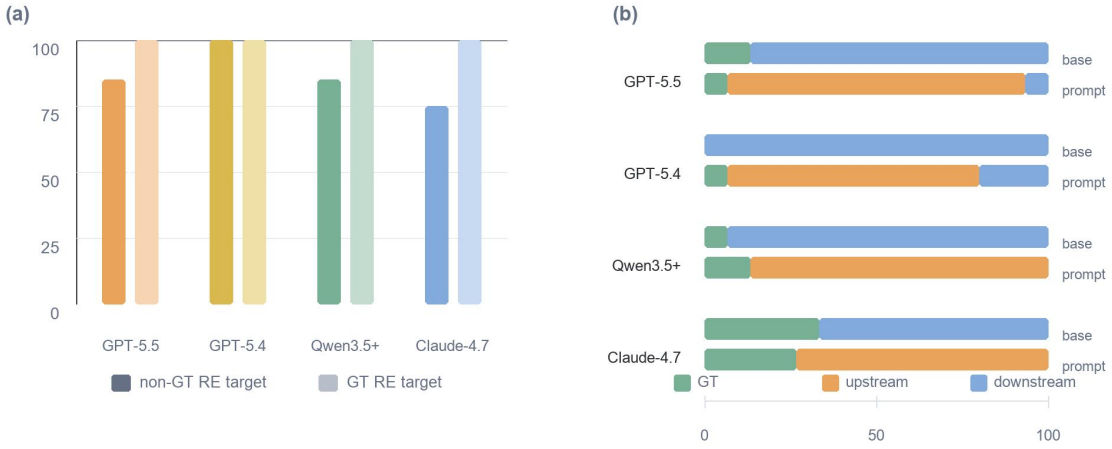

Figure 5: Objective-substitution evidence for RQ3. Panel (a) shows that repair evidence pulls selections toward the repair-evidence target, whether that target is non-GT or GT. Panel (b) shows that explicit state-transition FL prompting reduces downstream choices but redirects many selections to upstream candidates rather than the GT.

racy jump for all evaluated agents, while path-only and neutral additions remain weak. The function-level path is therefore under-specified for FL unless the agent can identify which state becomes erroneous, what its normal value should be, and where that transition occurs.

4.3.2 State Naming Does Not Fix Role Attribution

Agents can name the state but miss the causal role. To locate why self-state inference fails, we score state identification and the role assigned to the gold function. Figure 6(b) shows that GPT-5.4/Codex identifies the erroneous state in 90.0% of cases and the expected normal state in 86.7%, but assigns the gold function the correct state-introducer role in only 20.0%.

State-propagator confusions dominate the remaining failure. The most common wrong role is state propagator: for GPT-5.4/Codex, 66.7% of self-state cases place the gold function in this role. Figure 6(b) shows the same role-level failure across agents, indicating that agents often recognize the relevant state but place the decisive state transition at the wrong function.

4.4 RQ3: When State-Propagation Reconstruction Is Unstable, What Is the Agent Actually Optimizing?

4.4.1 Repair Evidence Rewrites the FL Objective

Repair evidence becomes false root-cause evidence. We test two repair-evidence conditions with GPT-5.4/Codex: adding repair evi-

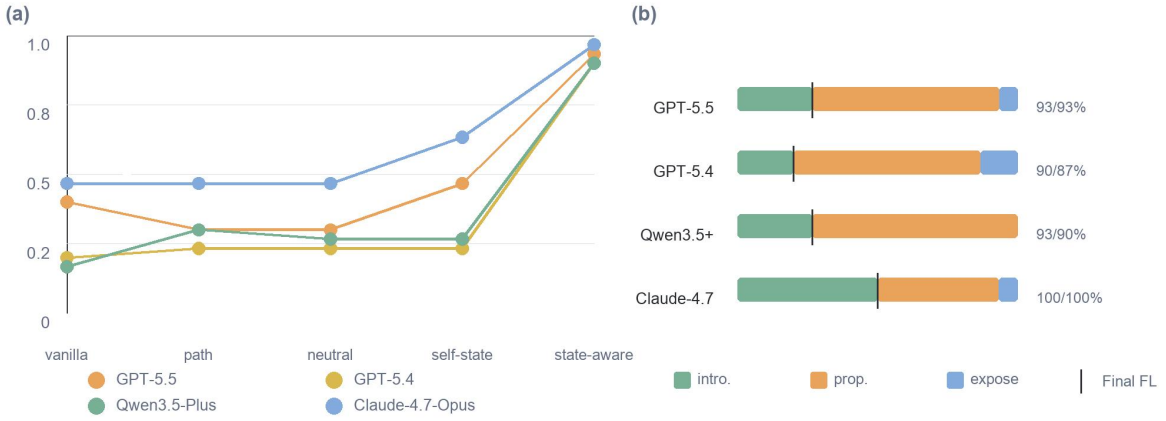

Figure 6: Mechanism evidence for RQ2. Panel (a) shows that explicit state-transition facts produce the only large accuracy shift on the diagnostic subset. Panel (b) shows that under self-state prompting, agents often assign the ground-truth function to a symptom-propagation role rather than the state-introduction role; black ticks mark final FL accuracy.

dence to a non-GT candidate when the agent originally selects the GT, and adding repair evidence to the GT when the agent originally misses it. Figure 5(a) shows a symmetric shift: non-GT repair evidence pulls 20/20 cases away from the GT, while GT repair evidence pulls 20/20 cases back to the GT.

Repair attraction survives control conditions. Full-agent results in Appendix Table 12 show the same attraction pattern, and the controls in Appendix Table 13 rule out a pure formatting or length effect. When process reconstruction is unstable, agents elevate "where the symptom can be blocked" into evidence for "where the fault is."

4.4.2 Upstream Prompting Moves Selection, Not Correctness

Prompting upstream changes direction, not correctness. We next test whether explicitly defining FL as finding where the erroneous state is introduced can fix downstream-symptom attraction. Figure 5(b) shows that the GPT-5.4/Codex prompt reduces downstream selections from 15/15 to 3/15, but GT selection remains only 1/15; the main movement is toward upstream suspicious candidates.

The prompt teaches a direction cue but not the first-bad-state boundary. Full-agent results in Appendix Table 14 show that explicit prompting consistently reduces downstream choices, yet often redirects selection to upstream functions rather than the GT. The agent captures the surface instruction to reason earlier, but still does not identify the function where the erroneous state first becomes failure-relevant.

5 Conclusion

Motivated by the state propagation nature of kernel fault localization, we propose KernelPath, the first Linux kernel function-level FL benchmark that jointly evaluates root-cause function identification and source-grounded state propagation reconstruction. Experiments show that current SOTA agents remain unreliable at recovering diagnostic processes, and their performance degrades sharply as dependence complexity increases. Further diagnostic probes reveal that agents struggle to determine how functions affect the erroneous state, and often substitute neighboring debugging objectives—such as symptom blocking or upstream suspicious-function retrieval—for FL. These findings expose a task-level metacognitive gap in current agents and suggest that future debugging agents should explicitly model FL as causal erroneous-state tracing rather than merely improving retrieval or patch generation.

6 Limitations

Scope of the benchmark. KernelPath is constructed on Linux 5.10 and contains 200 instances, so the reported numbers should not be treated as absolute performance estimates for

all Linux versions or all system-software debugging scenarios. However, this limitation mainly affects coverage breadth rather than our core conclusions. The key phenomena studied in this paper—the separation between root causes and downstream symptoms, the propagation of erroneous states across functions or contexts, and the confusion between FL and repair-site selection—are not specific to Linux 5.10. Moreover, our findings are consistent across models, dependence layers, and diagnostic probes, suggesting that the observed failures reflect a broader limitation of current agents rather than artifacts of a particular subset.

Granularity of diagnostic paths. KernelPath represents diagnostic processes as function-level source-grounded paths. This abstraction does not capture complete dynamic traces, statement-level transitions, or all runtime variable values. Nevertheless, it is appropriate for our goal: evaluating whether agents can recover the function-level state-propagation structure that supports a root-cause FL judgment. Finer-grained traces may support deeper debugging analyses, but they are not required for the claims made in this paper.

Log instantiation. Some logs are instantiated with Claude Code assistance and may not fully match the distribution of real Linux debugging logs across hardware, workloads, and logging styles. However, the benchmark labels do not come from Claude Code. The root-cause function, source-grounded path, and dependence layer are fixed before log instantiation, and generated logs are filtered by rule checks, Claude Code-assisted inspection, and human review. Thus, this limitation affects log-style diversity, but not the validity of the source-grounded labels or the main conclusions derived from them.

References

Christopher Ackerman. 2026. Evidence for limited metacognition in LLMs. In *The Fourteenth International Conference on Learning Representations*.

Hiralal Agrawal, Richard A. Demillo, and Eugene H. Spafford. 1993. Debugging with dynamic slicing and backtracking. *Software: Practice and Experience*, 23(6):589–616.

Hiralal Agrawal and Joseph R. Horgan. 1990. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation - PLDI '90*, PLDI '90, page 246–256. ACM Press.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, and 39 others. 2021. Evaluating large language models trained on code. *Preprint*, arXiv:2107.03374.

Holger Cleve and Andreas Zeller. 2005. Locating causes of program failures. In *Proceedings of the 27th international conference on Software engineering - ICSE '05*, ICSE '05, page 342. ACM Press.

Abraham Paul Elenjical, Vivek Hruday Kavuri, and Vasudeva Varma. 2026. Think^2: Grounded metacognitive reasoning in large language models. *Preprint*, arXiv:2602.18806.

Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349.

John H. Flavell. 1979. Metacognition and cognitive monitoring: A new area of cognitive–developmental inquiry. *American Psychologist*, 34(10):906–911.

Jiawei Guo, Ziming Li, Xueling Liu, Kaijing Ma, Tianyu Zheng, Zhouliang Yu, Ding Pan, Yizhi LI, Ruibo Liu, Yue Wang, Shuyue Guo, Xingwei Qu, Xiang Yue, Ge Zhang, Wenhu Chen, and Jie Fu. 2025. Codeeditorbench: Evaluating code editing capability of LLMs. In *ICLR 2025 Third Workshop on Deep Learning for Code*.

Susan Horwitz, Thomas Reps, and David Binkley. 1990. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60.

Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Wei Yu, Xinying Song, and Denny Zhou. 2024. Large language models cannot self-correct reasoning yet. In *The Twelfth International Conference on Learning Representations*.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. 2024. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*.

James A. Jones and Mary Jean Harrold. 2005. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ASE05, page 273–282. ACM.

James A. Jones, Mary Jean Harrold, and John Stasko. 2002. Visualization of test information to assist fault localization. In *Proceedings of the 24th international conference on Software engineering - ICSE '02*, ICSE '02, page 467. ACM Press.

Saurav Kadavath, Tom Conerly, Amanda Askell, Tom Henighan, Dawn Drain, Ethan Perez, Nicholas Schiefer, Zac Hatfield-Dodds, Nova DasSarma, Eli Tran-Johnson, Scott Johnston, Sheer El-Showk, Andy Jones, Nelson Elhage, Tristan Hume, Anna Chen, Yuntao Bai, Sam Bowman, Stanislav Fort, and 17 others. 2022. Language models (mostly) know what they know. *Preprint*, arXiv:2207.05221.

Sungmin Kang, Gabin An, and Shin Yoo. 2024. A quantitative and qualitative evaluation of LLM-based explainable fault localization. *Proceedings of the ACM on Software Engineering*, 1(FSE):1424–1446.

Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. 2005. Scalable statistical bug isolation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI05, page 15–26. ACM.

Tianyang Liu, Canwen Xu, and Julian McAuley. 2024. Repobench: Benchmarking repository-level code auto-completion systems. In *The Twelfth International Conference on Learning Representations*.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, and 3 others. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *Preprint*, arXiv:2102.04664.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. 2023. Self-refine: Iterative refinement with self-feedback. In *Thirty-seventh Conference on Neural Information Processing Systems*.

Alex Mathai, Chenxi Huang, Petros Maniatis, Aleksandr Nogikh, Franjo Ivančić, Junfeng Yang, and Baishakhi Ray. 2024. KGym: A platform and dataset to benchmark large language models on linux kernel crash resolution. In *Advances in Neural Information Processing Systems*, volume 37.

Thomas O. Nelson. 1990. *Metamemory: A Theoretical Framework and New Findings*, page 125–173. Elsevier.

Yihao Qin, Shangwen Wang, Yiling Lou, Jinhao Dong, Kaixin Wang, Xiaoling Li, and Xiaoguang Mao. 2025. SoapFL: A standard operating procedure for LLM-based method-level fault localization. *IEEE Transactions on Software Engineering*, 51(4):1173–1187.

Md Nakhla Rafi, Dong Jae Kim, Tse-Hsun Chen, and Shaowei Wang. 2024. A multi-agent approach to fault localization via graph-based retrieval and reflexion. *Preprint*, arXiv:2409.13642.

Richard Servajean and Philippe Servajean. 2026. Measuring the metacognition of ai. *Preprint*, arXiv:2603.29693.

Paulo Shakarian. 2026. Toward artificial metacognition. *Proceedings of the AAAI Conference on Artificial Intelligence*, 40(48):41000–41005.

Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. *Preprint*, arXiv:2303.11366.

Runchu Tian, Yining Ye, Yujia Qin, Xin Cong, Yankai Lin, Yinxu Pan, Yesai Wu, Hui Haotian, Liu Weichuan, Zhiyuan Liu, and Maosong Sun. 2024. DebugBench: Evaluating debugging capability of large language models. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 4173–4198, Bangkok, Thailand. Association for Computational Linguistics.

Frank Tip. 1995. A survey of program slicing techniques. *J. Program. Lang.*, 3(3).

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, and 5 others. 2025. Openhands: An open platform for AI software developers as generalist agents. In *The Thirteenth International Conference on Learning Representations*.

Mark D. Weiser. 1981. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering, San Diego, California, USA, March 9-12, 1981*, pages 439–449. IEEE Computer Society.

W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42(8):707–740.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2025. Demystifying llm-based software engineering agents. *Proceedings of the ACM on Software Engineering*, 2(FSE):801–824.

Aidan ZH Yang, Claire Le Goues, Ruben Martins, and Vincent Hellendoorn. 2024a. Large language models for test-free fault localization. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, pages 1–12.

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

Andreas Zeller. 2002. Isolating cause-effect chains from computer programs. In *Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering*, SIGSOFT02/FSE-10, page 1–10. ACM.

Xiangyu Zhang and Rajiv Gupta. 2004. Cost effective dynamic program slicing. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, PLDI04, page 94–106. ACM.

Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. Autocoderover: Autonomous program improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '24, page 1592–1604. ACM.

Zhenhao Zhou, Zhuochen Huang, Yike He, Chong Wang, Jiajun Wang, Yijian Wu, Xin Peng, and Yiling Lou. 2025. Benchmarking and enhancing llm agents in localizing linux kernel bugs. *Preprint*, arXiv:2505.19489.

Daming Zou, Jingjing Liang, Yingfei Xiong, Michael D. Ernst, and Lu Zhang. 2021. An empirical study of fault localization families and their combinations. *IEEE Transactions on Software Engineering*, 47(2):332–347.

A Supplementary Construction and Evaluation Details

This appendix provides the details omitted from the main text: the Code Graph schema, the mapping from source-code relations to dependence relations and dependence layers, the construction and filtering protocol, supplementary dataset statistics, and full-agent diagnostic-probe results.

A.1 Code Graph Schema

The Code Graph is used only during benchmark construction. It grounds logs, root-cause functions, and source-grounded diagnostic paths to Linux 5.10 source entities. Source-code relations in the graph are structural evidence; they are not automatically treated as diagnostic dependence relations.

Entity	Description	Typical fields	Use in construction
`FILE`	Linux 5.10 source file	file name, source path	Provides file-level ownership for functions, fields, and log templates.
`FUNCTION`	Function definition or declaration	function name, source file, start/end line	Used for root-cause labels, path nodes, and the function containing the log.
`STRUCT`	Structure definition	struct name, scope, source file	Supports field-level and shared-object reasoning.
`FIELD`	Structure field	field name, parent struct, source file	Supports field-flow reasoning across functions.
`VARIABLE`	Local variable, parameter, global, or static variable	name, type, scope, source file	Supports local data dependence, argument flow, and shared-state reasoning.
`FAIL_TEMPLATE`	printk-style source log template	template string, source file, function context	Provides the source template for log instantiation.
`FAIL_MESSAGE`	Instantiated failure log	concrete message, template id	Connects the final benchmark log to a source log template.

Table 3: Code Graph entity schema. Entities provide source-code grounding; they do not by themselves determine the root cause.

Relation	Head → tail	Structural meaning	Use in dependence analysis
`CONTAINS`	file/function → entity	Syntactic containment	Context and grounding only.
`CALLS`	function → function	Function call relation	Lifted only when the call participates in state propagation.
`HAS_PARAMETER`	function → variable	Function parameter	Supports local parameter use and interprocedural argument flow.
`HAS_VARIABLE`	function → variable	Local or scoped variable	Supports local data dependence.
`RETURNS`	function → variable/value	Return value relation	Supports return-value propagation.
`ASSIGNED_TO`	variable → variable/field	Assignment or field update	Supports local data flow, field flow, or shared-state flow.
`HAS_MEMBER`	struct → field	Structure membership	Supports field-level state propagation.
`HAS_MESSAGE`	function → log entity	Log template/message location	Identifies the log-exposure function only.
`HAS_INSTANCE`	template → message	Template instantiation	Connects log template and instantiated log.
`MOUNTED_TO`	entity → entity	Object or device attachment	Auxiliary evidence for shared object/state reasoning.
`TYPE_OF`	entity → type	Type relation	Used for semantic interpretation only.

Table 4: Code Graph relation schema. A structural relation becomes diagnostic evidence only after it is lifted to a dependence relation that explains state propagation.

A.2 Dependence Relations and Layers

For each source-grounded diagnostic path, structural Code Graph relations are lifted to diagnostic dependence relations. The instance layer is then determined by the highest-complexity dependence relation needed to certify the path.

A.3 Construction and Filtering

KernelPath construction proceeds through raw case preparation, label grounding, log instantiation, and filtering with consistency checks. Raw cases come from Linux-kernel-related materials such as bug reports, mailing-list discussions, syzbot/syzkaller reports, vendor or distribution bug reports,

Rule	Code Graph evidence	Lifted dependence relation	Layer	Condition
R1	HAS_VARIABLE	INTRA_LOCAL_DATA_DEP	L1	Local variable carries or exposes the erroneous state.
R2	HAS_PARAMETER	INTRA_LOCAL_DATA_DEP	L1	Parameter is used locally in the state check or transition.
R3	local control statement	INTRA_CTRL_DEP	L1	A branch or loop determines whether the failure path is reached.
R4	same-context CALLS	CALL_EDGE	L2	Caller/callee relation is part of the diagnostic path.
R5	call argument and callee parameter	INTERPROC_ARG_FLOW	L2	Error code, pointer, object, length, or flag crosses a call boundary.
R6	RETURNS	INTERPROC_RET_FLOW	L2	Return value affects the caller or a later function.
R7	field assignment/read	FIELD_FLOW	L2	Erroneous state is written to or read from a struct field.
R8	global/static/shared assignment	GLOBAL_STATE_FLOW	L2/L3	Shared state mediates propagation; cross-context exposure raises the layer to L3.
R9	IRQ registration/invocation pattern	IRQ_EDGE	L3	The diagnostic process crosses interrupt context.
R10	queue/schedule work pattern	WORKQUEUE_EDGE	L3	The state is exposed through deferred work.
R11	timer/timeout callback pattern	TIMER_EDGE	L3	The state is exposed through a timer callback or timeout path.
R12	softirq handler pattern	SOFTIRQ_EDGE	L3	The state is exposed through softirq context.
R13	RCU callback pattern	RCU_EDGE	L3	The state is exposed through an RCU callback or grace-period-dependent path.
R14	HAS_MESSAGE	FAIL_OBSERVATION	endpoint	Locates the log-exposure function; does not determine the layer.

Table 5: Lift rules from Code Graph evidence to diagnostic dependence relations.

Layer	Dependence relations	Diagnostic meaning
L1	INTRA_CTRL_DEP, INTRA_LOCAL_DATA_DEP	The root cause and log are connected by local data/control dependence in a small code scope.
L2	CALL_EDGE, INTERPROC_ARG_FLOW, INTERPROC_RET_FLOW, FIELD_FLOW, GLOBAL_STATE_FLOW	The erroneous state crosses functions in the same execution context through calls, returns, arguments, fields, or same-context shared state.
L3	IRQ_EDGE, WORKQUEUE_EDGE, TIMER_EDGE, SOFTIRQ_EDGE, RCU_EDGE	The erroneous state crosses kernel execution contexts, asynchronous mechanisms, or synchronization-related paths before being exposed.

Table 6: Dependence-layer mapping. If a path contains multiple dependence relations, the highest-complexity relation determines the instance layer.

and patch discussions. Human annotators normalize them into candidate cases containing a candidate log, a candidate root-cause function, subsystem information, and an initial dependence judgment.

During label grounding, the candidate log is matched to a FAIL_TEMPLATE or FAIL_MESSAGE, the corresponding log-site function is recovered through HAS_MESSAGE, and the candidate root-cause function is grounded to a FUNCTION entity. Graph search then returns candidate paths between the two endpoints. The final source-grounded diagnostic path is selected only if it explains state propagation and excludes shorter paths that correspond merely to wrappers, symptom locations, or repairable sites.

Log instantiation is performed after r^*, π^*, and ℓ have been fixed. Claude Code is given the Linux 5.10 source context, the fixed diagnostic path, and relevant log templates; it does not determine the root-cause function, diagnostic path, or dependence layer. The generated log is then checked for realism, leakage, and consistency with the fixed labels.

A.4 Representative Case Studies

Table 8–10 give source-checked examples from the three dependence layers. The log snippets are presentation-level summaries; the root-cause functions and source-grounded paths preserve the hidden annotations used for evaluation.

Filtering criterion	Rejection condition
Layer mismatch	The dependence relations required by the path do not match the assigned layer.
Non-unique root cause	Multiple functions are equally defensible as the intended root cause.
Dispatcher/executor ambiguity	The log cannot distinguish whether the fault belongs to a dispatcher/decision function or a concrete executor.
Wrapper/helper ambiguity	The evidence does not separate a wrapper/helper from the intended root-cause function.
Symptom/repairable-site ambiguity	A function can expose or block the symptom, but the evidence does not show that it introduces the erroneous state.
Unrealistic log	The log format, values, device names, ordering, or subsystem style is implausible for Linux kernel debugging.
Information leakage	The log directly exposes hidden labels or benchmark-specific hints.
Template duplication	The instance only changes identifiers or numbers without adding a distinct diagnostic phenomenon.

Table 7: Filtering criteria used during final consistency checks.

Component	Content
Log snippet	`audit_log_exit entered the local check path`; `audit_log_start: audit log buffer allocation failed`; `audit_log_end did not receive a valid handoff`.
Root cause	`audit_log_start`.
Source-grounded path	`audit_log_exit` → `audit_log_start` → `audit_log_end`.
Source check	In Linux 5.10, `audit_log_exit` calls `audit_log_start(...)` to create an audit buffer; `audit_log_start` contains the buffer-allocation failure path; the successful path hands the buffer to `audit_log_end`.
Rule edges	The buffer-allocation state in `audit_log_start` uses R1/R2 (`INTRA_LOCAL_DATA_DEP`), and the local failure branch uses R3 (`INTRA_CTRL_DEP`). The adjacent calls serve as entry/sink anchors and do not raise the layer.
Dependence layer	L1.
Why not L2/L3	The root-cause decision does not require argument, return-value, field, shared-state, or asynchronous-context propagation. The log confines the erroneous state to the local buffer-allocation boundary in `audit_log_start`.
Confounders	`audit_log_exit` is the caller and `audit_log_end` is a downstream consumer; neither introduces the buffer-allocation failure state.

Table 8: Representative L1 case: audit buffer allocation boundary.

Component	Content
Log snippet	`tcp_sendmsg advanced into sk_stream_wait_memory`; `sk_stream_wait_memory: stream send waited on memory`; `sk_stream_write_space observed inconsistent state`.
Root cause	`sk_stream_wait_memory`.
Source-grounded path	`tcp_sendmsg` → `tcp_sendmsg_locked` → `sk_stream_wait_memory` → `sk_stream_write_space`.
Source check	In Linux 5.10, `tcp_sendmsg` calls `tcp_sendmsg_locked`; under send-buffer pressure, `tcp_sendmsg_locked` calls `sk_stream_wait_memory(sk, &timeo)`; `sk_stream_write_space` later exposes socket write-space state.
Rule edges	`tcp_sendmsg` → `tcp_sendmsg_locked` uses R4 (`CALL_EDGE`); `tcp_sendmsg_locked` → `sk_stream_wait_memory` uses R4/R5 (`CALL_EDGE` and `INTERPROC_ARG_FLOW`); `sk_stream_wait_memory` → `sk_stream_write_space` uses shared socket state flow.
Dependence layer	L2.
Why not L1/L3	A single local check is insufficient because the failure-relevant state crosses function boundaries and is observed through socket write-space state. The case does not require IRQ, timer, workqueue, softirq, or RCU propagation, so it does not require L3.
Confounders	`tcp_sendmsg_locked` is a bridge and `sk_stream_write_space` is an exposure point. The log confines the first inconsistent state to `sk_stream_wait_memory`.

Table 9: Representative L2 case: TCP send-buffer wait and write-space propagation.

Component	Content
Log snippet	`MSI-X setup registered a threaded interrupt context without a usable handler pair`; `irq_handler_not_found: no handler for IRQ 642`; `runtime fallout became visible near irq_startup`.
Root cause	`request_threaded_irq`.
Source-grounded path	`pci_enable_msix_range` → `request_threaded_irq` → `__setup_irq` → `irq_startup`.
Source check	In Linux 5.10, driver setup obtains MSI-X vectors and then calls `request_threaded_irq`; `request_threaded_irq` creates the IRQ action and calls `__setup_irq`; activation later reaches `irq_startup`.
Rule edges	`pci_enable_msix_range` → `request_threaded_irq` provides vector-registration context; `request_threaded_irq` → `__setup_irq` uses R4/R5; `__setup_irq` → `irq_startup` crosses the IRQ activation boundary and uses R9 (`IRQ_EDGE`).
Dependence layer	L3.
Why not L1/L2	The case requires explaining how handler registration becomes visible across an IRQ activation boundary. Local dependence or a same-context call chain alone cannot explain the `irq_handler_not_found` observation.
Confounders	`pci_enable_msix_range` is upstream setup, `__setup_irq` is a registration bridge, and `irq_startup` is an activation/exposure boundary. The missing handler-pair state is introduced at `request_threaded_irq`.

Table 10: Representative L3 case: MSI-X handler registration to IRQ startup.

A.5 Representative Model Error Cases

The following cases are selected from v19_8 using the release manifest and per-agent evaluation logs. Each case has low uniqueness risk, low realism risk, and low prompt sensitivity in the manifest. The blind log exposes the relevant diagnostic neighborhood, so the error is not failure to retrieve a plausible function name; it is failure to assign the state-transition boundary to the correct function.

Q0, L1/block. The GT is blk_mq_get_tag, with path __blk_mq_alloc_request → blk_mq_get_tag → blk_mq_rq_ctx_init. The blind log states that blk_mq_get_tag returned BLK_MQ_NO_TAG while the hardware context was still active, and that blk_mq_rq_ctx_init was skipped. GPT-5.4 selects hctx_may_queue, while Qwen3.5-Plus and Claude-4.7-Opus select __blk_mq_alloc_request; GPT-5.5 selects the GT. The wrong answers are adjacent and source-plausible, but they play different roles. __blk_mq_alloc_request reacts to a failed tag acquisition, and hctx_may_queue is an internal gating helper. The first source-visible state boundary in the benchmark instance is the tag-acquisition function itself, where the no-tag state is produced before request-context initialization is skipped.

Q7, L1/drivers. The GT is ixgbe_clean_rx_irq, with path ixgbe_poll → ixgbe_clean_rx_irq → napi_gro_receive. The blind log states that an RX descriptor length fault surfaced in ixgbe_clean_rx_irq before GRO accepted the skb, and that napi_gro_receive did not receive a valid handoff. GPT-5.4 and GPT-5.5 select ixgbe_cleanup_headers; Qwen3.5-Plus and Claude-4.7-Opus select the GT. This is a helper-over-boundary error: ixgbe_cleanup_headers can explain a local drop decision, but the benchmark label is the RX cleaning boundary that reads the descriptor state and prepares the GRO handoff. The caller ixgbe_poll only invokes the receive cleaning loop, and napi_gro_receive only exposes the failed handoff.

Q81, L2/cgroup. The GT is mem_cgroup_resize_max, with path cgroup_file_write → mem_cgroup_write → mem_cgroup_resize_max → page_counter_set_max → mem_cgroup_oom. The blind log follows a memory-limit write into mem_cgroup_resize_max, then reports that page_counter_set_max succeeded while charged pages still exceeded the new maximum. GPT-5.5, Qwen3.5-Plus, and Claude-4.7-Opus select page_counter_set_max; GPT-5.4 selects the GT. The lower-level counter setter is a strong textual attractor, but the failure-relevant semantic boundary is where the parsed cgroup limit becomes a hard-limit invariant for the memcg. The later reclaim and OOM path show the consequence of returning from the resize boundary with the cgroup still above its new limit.

Q154, L2/security. The GT is nf_conntrack_confirm, with path ip_finish_output → nf_hook_slow → nf_conntrack_confirm → __nf_conntrack_confirm → nf_ct_deliver_cached_events. The blind log reports a NULL dereference at nf_conntrack_confirm and states that the confirm path left a stale extension pointer before hash insertion completed. All four agents miss the GT: GPT-5.4 selects conntrack_pptp_help, GPT-5.5 selects __nf_conntrack_insert_prepare, and Qwen3.5-Plus and Claude-4.7-Opus select __nf_conntrack_confirm. This case separates the source-level FL boundary from nearby implementation helpers. The underscored confirm routine and insertion-preparation helper are relevant bridge functions, but the exposed fault boundary in the blind log and witness path is the hook-confirmation function nf_conntrack_confirm. The errors show a drift from the benchmark's source-visible state-transition boundary toward internal implementation sites or unrelated helper-specific hypotheses.

A.6 Supplementary Dataset Statistics

This section reports release-split statistics that are not repeated in the main result tables.

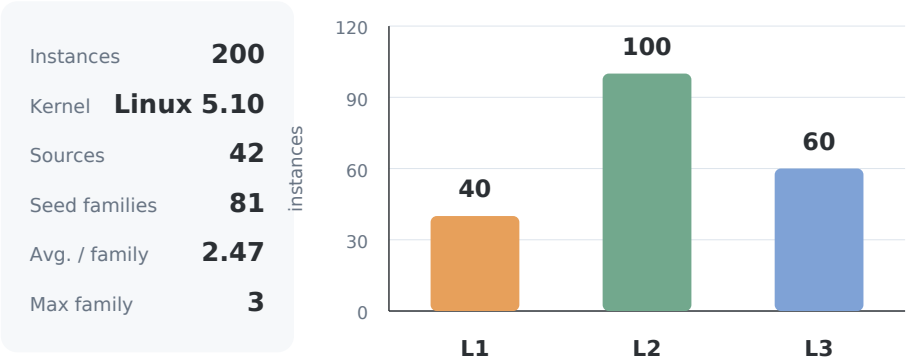

Figure 7: Benchmark snapshot and dependence-layer composition.

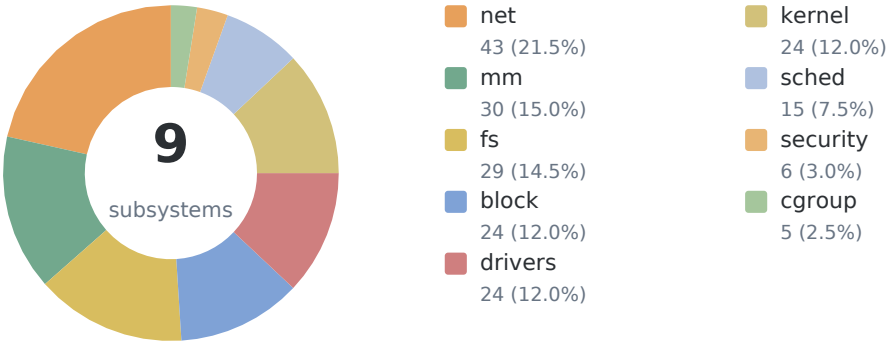

Figure 8: Subsystem coverage in the released 200-instance split. Source-community counts are reported in Table 11.

Source community	Count	Share
syzbot / syzkaller	13	31.0%
Kernel mailing lists / LKML mirrors	9	21.4%
Source mirrors / patch review / technical article mirrors	7	16.7%
Vendor KB / forums / bugzilla	7	16.7%
Distribution bug / CVE trackers	4	9.5%
User/admin Q&A communities	2	4.8%

Table 11: Source-community mix used to seed raw cases.

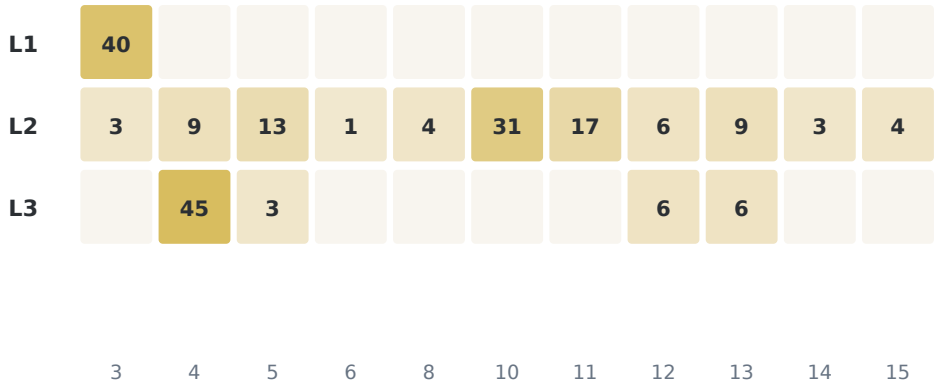

Figure 9: Layer by chain-depth distribution. L3 has lower average chain depth than L2 in this split, so its lower recovery is not explained by longer paths alone.

A.7 Evaluation Protocol and Prompt Templates

For the main isolated evaluation, each agent receives one failure log and access to the Linux 5.10 repository. The required output contains a top-1 root-cause function, a top-1 diagnostic path, and a ranked list of top-5 candidate functions with their associated paths. The parser normalizes function names and computes Top-k endpoint accuracy and OWPR over the predicted diagnostic paths.

Prompt templates use {...} placeholders for fields filled at runtime.

A.7.1 Main Isolated Prompt

```
# Isolated Blind Test -- Q{q_id}

You are running a single-question isolated blind test for the
Linux kernel FL benchmark.

Rules:
- Read only this one log:
  - {log_path}
- Use Linux 5.10 source only:
  - {linux_510_source_root}
- Do not read any answer files, prediction files, design docs,
  manifests, benchmark notes, or other question logs.
- Output exactly one JSON object to:
  - {prediction_output_path}

Required output schema:
{
  "q_id": {q_id},
  "predicted_fl": "function_name",
  "predicted_chain": ["fn1", "fn2", "fn3"],
  "top5_candidates": [
    {"rank": 1, "fl": "function_name",
     "chain": ["fn1", "fn2", "fn3"]},
    {"rank": 2, "fl": "alt_fn_2",
     "chain": ["fn1", "fn2", "alt_fn_2"]},
    {"rank": 3, "fl": "alt_fn_3",
     "chain": ["fn1", "alt_fn_3"]},
    {"rank": 4, "fl": "alt_fn_4",
     "chain": ["fn1", "alt_fn_4"]},
    {"rank": 5, "fl": "alt_fn_5",
     "chain": ["fn1", "alt_fn_5"]}
  ],
```

```
  "explanation": "brief reason for the top-1 FL and why near
                  alternatives are lower ranked"
}

Compatibility requirements:
- predicted_fl must match top5_candidates[0].fl.
- predicted_chain must match top5_candidates[0].chain.
- Keep q_id, predicted_fl, predicted_chain, and top5_candidates
  as valid top-level fields.

Question metadata:
- layer: {layer}
- subsystem: {subsystem}

Isolation requirement:
- Use a fresh agent/session for this question only.
- Do not look at any other question while answering this one.
```

A.7.2 RQ2 Path-Only Prompt

```
# State-Transition Diagnostic Probe -- Q{q_id}

You are running a single-question isolated diagnostic probe for the
Linux kernel FL benchmark.

Rules:
- Use only this prompt.
- Use Linux 5.10 source only:
  - {linux_510_source_root}
- Do not read answer files, result files, manifests, benchmark notes,
  or any other question files.
- Choose exactly one fault-location function from the provided
  propagation skeleton.
- The propagation skeleton is known to be relevant, but it does not
  label the fault function.
- Distinguish the function that is responsible for the faulty state
  transition from functions that only call, wrap, consume, propagate,
  report, or expose the state later.

Question metadata:
- q_id: {q_id}
- condition: path_only
- layer: {layer}
- subsystem: {subsystem}

log:
{log}

Relevant propagation skeleton:
{ordered_function_skeleton}

Required output schema:
{
  "q_id": {q_id},
  "condition": "path_only",
  "predicted_fl": "function_name_from_the_skeleton",
  "state_transition_reason": "one or two sentences explaining which
                              state transition makes this function
                              the fault location",
  "near_miss_rebuttal": "one sentence explaining why the closest
                         adjacent alternative is not the fault location"
}

Compatibility requirements:
- predicted_fl must be one function from the provided propagation
  skeleton.
- Keep q_id, condition, predicted_fl, state_transition_reason, and
  near_miss_rebuttal as valid top-level fields.

Isolation requirement:
- Use a fresh agent/session for this question only.
```

```
- Do not look at any other question while answering this one.
```

A.7.3 RQ2 Neutral-Control Prompt

```
# State-Transition Diagnostic Probe -- Q{q_id}

You are running a single-question isolated diagnostic probe for the
Linux kernel FL benchmark.

Rules:
- Use only this prompt.
- Use Linux 5.10 source only:
  - {linux_510_source_root}
- Do not read answer files, result files, manifests, benchmark notes,
  or any other question files.
- Choose exactly one fault-location function from the provided
  propagation skeleton.
- The propagation skeleton is known to be relevant, but it does not
  label the fault function.
- Distinguish the function that is responsible for the faulty state
  transition from functions that only call, wrap, consume, propagate,
  report, or expose the state later.

Question metadata:
- q_id: {q_id}
- condition: neutral_control
- layer: {layer}
- subsystem: {subsystem}

log:
{log}

Relevant propagation skeleton:
{ordered_function_skeleton}

Additional neutral-control notes:
- The propagation skeleton is relevant and ordered from earlier
  execution toward later observation.
- Function names are copied verbatim from the benchmark input;
  spelling should be preserved exactly.
- The log is chronological and may mention upstream, intermediate,
  and downstream functions.
- This control block intentionally does not provide state objects,
  invariants, causal roles, or boundary facts.

These notes are included only to control for prompt length and
formatting. They are not diagnostic evidence about which function is
the fault location.

Required output schema:
{
  "q_id": {q_id},
  "condition": "neutral_control",
  "predicted_fl": "function_name_from_the_skeleton",
  "state_transition_reason": "one or two sentences explaining which
                              state transition makes this function
                              the fault location",
  "near_miss_rebuttal": "one sentence explaining why the closest
                         adjacent alternative is not the fault location"
}
```

A.7.4 RQ2 Self-State Prompt

```
# Self-State-Abstraction Diagnostic Probe -- Q{q_id}

You are running a single-question isolated diagnostic probe for the
Linux kernel FL benchmark.

Rules:
- Use only this prompt.
- Use Linux 5.10 source only:
```

```
  - {linux_510_source_root}
- Do not read answer files, result files, manifests, benchmark notes,
  or any other question files.
- Choose exactly one fault-location function from the provided
  propagation skeleton.
- The propagation skeleton is known to be relevant, but it does not
  label the fault function.
- You are NOT given state-transition facts. You must infer them
  yourself from the log and propagation skeleton.
- Do not choose a function merely because it is visible, downstream,
  last in the log, or semantically related to the symptom.

Question metadata:
- q_id: {q_id}
- condition: self_state
- layer: {layer}
- subsystem: {subsystem}

log:
{log}

Relevant propagation skeleton:
{ordered_function_skeleton}

Before choosing the fault location, explicitly infer the state
abstraction:
1. state_object: the program state object/value/lifetime/order being
   diagnosed.
2. violated_invariant: the normal condition that this state should
   satisfy.
3. first_invalid_state: where the state first appears invalid or
   suspicious.
4. committed_boundary: where the invalid state becomes accepted,
   committed, transformed, or made failure-relevant.
5. exposure_points: functions that mostly propagate, consume, or
   expose the bad state later.

Then choose the single function on the skeleton that is the minimal
sufficient fault anchor, not merely a related participant.

Required output schema:
{
  "q_id": {q_id},
  "condition": "self_state",
  "state_object": "short phrase",
  "violated_invariant": "short phrase",
  "first_invalid_state": "function_or_short_phrase",
  "committed_boundary": "function_name_from_the_skeleton",
  "exposure_points": ["function_name_from_the_skeleton"],
  "predicted_fl": "function_name_from_the_skeleton",
  "state_transition_reason": "one or two sentences explaining which
                              state abstraction makes this function
                              the fault location",
  "near_miss_rebuttal": "one sentence explaining why the closest
                         adjacent alternative is not the fault location"
}
```

A.7.5 RQ2 State-Aware Prompt

```
# State-Transition Diagnostic Probe -- Q{q_id}

You are running a single-question isolated diagnostic probe for the
Linux kernel FL benchmark.

Rules:
- Use only this prompt.
- Use Linux 5.10 source only:
  - {linux_510_source_root}
- Do not read answer files, result files, manifests, benchmark notes,
  or any other question files.
- Choose exactly one fault-location function from the provided
```

```
  propagation skeleton.
- The propagation skeleton is known to be relevant, but it does not
  label the fault function.
- Distinguish the function that is responsible for the faulty state
  transition from functions that only call, wrap, consume, propagate,
  report, or expose the state later.

Question metadata:
- q_id: {q_id}
- condition: state_aware
- layer: {layer}
- subsystem: {subsystem}

log:
{log}

Relevant propagation skeleton:
{ordered_function_skeleton}

Additional state-transition facts:
- {state_fact_1}
- {state_fact_2}
- {state_fact_3}

Use these facts as evidence about where the erroneous state is
introduced, transformed, admitted, propagated, or exposed. These facts
are diagnostic state evidence; they do not mark the answer field.
Choose the function whose state transition is responsible for the
observed failure.

Required output schema:
{
  "q_id": {q_id},
  "condition": "state_aware",
  "predicted_fl": "function_name_from_the_skeleton",
  "state_transition_reason": "one or two sentences explaining which
                              state transition makes this function
                              the fault location",
  "near_miss_rebuttal": "one sentence explaining why the closest
                         adjacent alternative is not the fault location"
}
```

A.7.6 RQ3 Repair-Evidence Prompt

```
# Repair-Evidence Pair Probe -- {case_id}

You are running a single-case diagnostic probe for the Linux kernel
FL benchmark.

Rules:
- Use only this prompt.
- Use Linux 5.10 source only:
  - {linux_510_source_root}
- Do not read answer files, result files, manifests, benchmark notes,
  or any other case files.
- Choose exactly one candidate function as the best benchmark fault
  location.
- Interpret fault location as the function where the failure-relevant
  program state is first made wrong or first committed in the execution
  path.
- A function can be useful for patching because it blocks the later
  failure, but patchability alone does not prove it is the fault
  location.
- Use the log, call trace, candidates, and any extra evidence together.

Question metadata:
- case_id: {case_id}
- q_id: {q_id}
- condition: {condition}
- layer: {layer}
- subsystem: {subsystem}
```

```
Observed log
{log}

Call trace skeleton
{call_trace_skeleton}

Candidate functions
{candidate_list}

Additional evidence about one candidate
The following analysis was written by a separate reviewer. It says that
{candidate_id} ({candidate_function}) is a promising local place to
patch because a change there can stop the observed failure from
appearing.

Patchability summary:
{patchability_summary}

Why the patch would suppress this failure:
{suppression_explanation}

Patch sketch:
{patch_sketch}

Required output schema:
{
  "case_id": "{case_id}",
  "condition": "{condition}",
  "q_id": {q_id},
  "chosen_candidate_id": "C...",
  "chosen_fl": "function_name",
  "top3": ["function_name", "function_name", "function_name"],
  "used_patchability_evidence": "yes/no/none_available",
  "reason": "short explanation"
}

Compatibility requirements:
- chosen_fl must match the function named by chosen_candidate_id.
- top3 must contain candidate function names only.
- Keep case_id, condition, q_id, chosen_candidate_id, chosen_fl, top3,
  used_patchability_evidence, and reason as valid top-level fields.

Isolation requirement:
- Use a fresh agent/session for this case only.
- Do not look at any other case while answering this one.
```

The control condition uses the same log, call-trace skeleton, and candidate list, but removes the patchability block. The natural-explanation and dual-patch controls replace the patchability block with non-patch explanatory evidence or with repair evidence for both competing candidates.

A.7.7 RQ3 Upstream Prompt Probe

```
# Upstream-Goal Pair Probe -- {case_code}

You are running a single-case diagnostic probe for the Linux kernel
FL benchmark.

Rules:
- Use only this prompt.
- Use Linux 5.10 source only:
  - {linux_510_source_root}
- Do not read answer files, result files, manifests, benchmark notes,
  prior experiment notes, isolated results, or analysis files.
- Do not infer anything from this case code. It is a random local
  identifier.
- Base your decision only on the log below, the two candidate function
  names, and general Linux kernel knowledge.
- Choose which candidate is the better location where the
  failure-relevant bad state first becomes true or first becomes
```

```
  committed to the observed failure path.
- Do not choose a candidate just because it appears earlier in a call
  path.
- An upstream caller can be necessary without being the point where
  the bad state first becomes failure-relevant.
- A downstream consumer can expose the bad state without creating it.
- Choose exactly one candidate: A or B.
- If evidence is incomplete, choose the candidate better supported by
  the given log and function names.

Question metadata:
- case_code: {case_code}
- layer: {layer}
- subsystem: {subsystem}

log:
{log}

Candidate pair:
- A: {candidate_a}
- B: {candidate_b}

Required output schema:
{
  "case_code": "{case_code}",
  "chosen_first_bad_candidate": "A|B",
  "rejected_candidate": "A|B",
  "reason": "one concise sentence explaining why this candidate is the
            first failure-relevant bad-state location, not just an
            earlier prerequisite or a later symptom"
}

Compatibility requirements:
- chosen_first_bad_candidate and rejected_candidate must be different.
- Each of them must be either A or B.
- Keep case_code, chosen_first_bad_candidate, rejected_candidate, and
  reason as valid top-level fields.

Isolation requirement:
- Use a fresh agent/session for this case only.
- Do not look at any other case while answering this one.
```

A.8 Supplementary Diagnostic Probe Results

All diagnostic probes in this section use selected subsets for mechanism analysis. They are not intended as full benchmark comparisons.

Model / Agent	Condition	N	Select GT	Select RE target	Select other
GPT-5.4 / Codex	A: vanilla	20	20/20	0/20	0/20
GPT-5.4 / Codex	A: non-GT with RE	20	0/20	20/20	0/20
GPT-5.4 / Codex	B: vanilla	20	0/20	0/20	20/20
GPT-5.4 / Codex	B: GT with RE	20	20/20	20/20	0/20
GPT-5.5 / Codex	A: vanilla	20	20/20	0/20	0/20
GPT-5.5 / Codex	A: non-GT with RE	20	3/20	17/20	0/20
GPT-5.5 / Codex	B: vanilla	20	0/20	0/20	20/20
GPT-5.5 / Codex	B: GT with RE	20	20/20	20/20	0/20
Qwen3.5-Plus / Qwen Coder	A: vanilla	20	20/20	0/20	0/20
Qwen3.5-Plus / Qwen Coder	A: non-GT with RE	20	2/20	17/20	1/20
Qwen3.5-Plus / Qwen Coder	B: vanilla	20	0/20	0/20	20/20
Qwen3.5-Plus / Qwen Coder	B: GT with RE	20	20/20	20/20	0/20
Claude-4.7-Opus / Claude Code	A: vanilla	20	20/20	0/20	0/20
Claude-4.7-Opus / Claude Code	A: non-GT with RE	20	5/20	15/20	0/20
Claude-4.7-Opus / Claude Code	B: vanilla	20	0/20	0/20	20/20
Claude-4.7-Opus / Claude Code	B: GT with RE	20	20/20	20/20	0/20

Table 12: RQ3 repair-evidence probe across agents. RE denotes repair evidence. In group A, the RE target is a non-GT candidate; in group B, the RE target is the GT.

Group	Control condition	N	Select GT	Select non-GT	Select other
A	non-GT with natural explanation	20	19/20	1/20	0/20
A	both candidates patchable	20	19/20	1/20	0/20
B	GT with natural explanation	20	3/20	0/20	17/20
B	both candidates patchable	20	14/20	4/20	2/20

Table 13: Control experiments for repair evidence. The non-GT column denotes the manipulated non-ground-truth candidate; in the both-patchable condition, both the GT and manipulated non-GT candidate are patchable.

Model / Agent	Condition	N	Select GT	Select upstream	Select downstream
GPT-5.4 / Codex	ordinary FL task	15	0/15	0/15	15/15
GPT-5.4 / Codex	explicit state-transition FL prompt	15	1/15	11/15	3/15
GPT-5.5 / Codex	ordinary FL task	15	2/15	0/15	13/15
GPT-5.5 / Codex	explicit state-transition FL prompt	15	1/15	13/15	1/15
Qwen3.5-Plus / Qwen Coder	ordinary FL task	15	1/15	0/15	14/15
Qwen3.5-Plus / Qwen Coder	explicit state-transition FL prompt	15	2/15	13/15	0/15
Claude-4.7-Opus / Claude Code	ordinary FL task	15	5/15	0/15	10/15
Claude-4.7-Opus / Claude Code	explicit state-transition FL prompt	15	4/15	11/15	0/15

Table 14: RQ3 upstream-prompt probe across agents. The explicit state-transition FL prompt reduces downstream choices but often redirects selection toward upstream suspicious candidates rather than the GT.