CIRDAN: STRUCTURED DEEP INVESTIGATION FOR LONG-HORIZON CODE DIAGNOSIS

Anonymous authors
Paper under double-blind review

ABSTRACT

Diagnosing distributed system failures requires long-horizon investigation: LLM agents must trace causality across components, connect observed symptoms to responsible code, and distinguish root causes from downstream symptoms. Through an empirical study, we observe that agents often settle on plausible explanations but leave key causal steps unexplored. We introduce CIRDAN, a framework for structured investigation of such failures. CIRDAN performs broad repository survey, triages competing hypotheses, and explores promising explanations in a tree that supports refinement and backtracking. Its context management retains detailed evidence along each branch and summaries of alternatives, while preserving shared prefixes for prompt-cache reuse. Tool-call context injection brings in structural and architectural information to help the agent look beyond the currently inspected subsystem. We evaluate CIRDAN against four agent frameworks across five models on 34 real-world bugs from 11 distributed systems. CIRDAN improves diagnosis accuracy with four of the five models. It raises block coverage by 6.3–22.7 percentage points with GPT-5.2-Codex and by 7.4 and 3.6 points with Opus 5 and Gemini-3.8-Flash. With GPT-6-Astra, it maintains coverage comparable to mini-SWE-agent's while reporting half as many code blocks, raising precision from 42.2% to 61.4%. Our results show that keeping competing explanations and their evidence available, while leaving investigation decisions to the model, improves code diagnosis with most models, and that the harness's budgets and guidance must match how each model reasons.

1 INTRODUCTION

Distributed systems underpin modern cloud services, yet their sheer scale and complexity make failures inevitable. Diagnosing these failures is notoriously challenging. Developers must interpret ambiguous symptoms and sift through vast fragmented logs and stack traces across numerous components to pinpoint the responsible code, often without a reliable reproducer (Yuan et al., 2010; Pan et al., 2024). Observability and fault-localization solutions have made it easier to trace execution and narrow the search (Mace et al., 2015; Zhang et al., 2019). Yet developers often still have to perform a time-consuming *last-mile reasoning* step: connect these clues to reconstruct the full causal chain and identify the specific buggy location at code level.

LLM agents offer a way to automate this reasoning. By interleaving reasoning and tool use (Yao et al., 2023b), they can navigate repositories, gather evidence, and resolve software issues (Yang et al., 2024; Wang et al., 2025). But can they sustain *long-horizon investigations* when the cause is unknown, evidence is incomplete, reproduction is limited, and multiple explanations remain plausible? This ability is essential for agents to take responsibility for complex system development and maintenance. We use distributed system diagnosis as a demanding test of this ability.

Our empirical study across coding agents and major proprietary and open-weight models identifies a recurring failure mode pattern: agents can trace the evidence supplied with an issue yet remain anchored to plausible, incomplete explanations, mistaking downstream mechanisms for causes. Once an initial explanation sets the investigation direction, each file read can add detail to that account without testing its assumptions. The run appears productive, yet continued reasoning reinforces the same incomplete diagnosis. §2 summarizes these findings.

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.

Our central insight is that an agent must be able to pursue a hypothesis in depth while keeping earlier alternatives available for revision. We introduce CIRDAN, a framework for structured deep investigation that organizes this process as a tree of competing explanations (§3). Each branch retains its evidence and reasoning history, allowing the agent to refine a hypothesis, backtrack, and resume an alternative without reconstructing its earlier work.

CIRDAN begins with a broad repository survey, then uses brief triage passes to gather evidence for competing hypotheses before investigating selected branches in depth. Compact summaries share findings across branches, while shared context prefixes support prompt-cache reuse. Context injection supplies structural and architectural information to help the agent discover relevant code beyond its initial search scope. Budgets and guidance adapt to each model's reasoning behavior while preserving this shared approach (§4.1).

We evaluate five models on 34 real-world issues from 11 distributed systems (§4). CIRDAN improves diagnosis accuracy with four of the five models. With GPT-5.2-Codex, it improves block coverage by 6.3–22.7 percentage points over the three baselines, and with Opus and Gemini, for which the harness was not tuned, by 7.4 and 3.6 points. With GPT-6-Astra, its coverage is comparable to mini-SWE-agent's (58.1% versus 60.8%) while it reports half as many code blocks, raising precision from 42.2% to 61.4%. These results show model-dependent gains in recovering and selecting responsible code, with different inference tradeoffs. §4.3 examines how investigations compare explanations and choose the code they report, and §4.4 reports small ablations of reasoning effort, branching, and multi-agent layouts. §5 relates our approach to coding-agent harnesses, hypothesis search, and diagnosis systems.

Our contributions are:

- An empirical study of investigative reasoning across agents and models, using distributed-system diagnosis to identify recurring failures in hypothesis exploration and revision.
- CIRDAN, a framework that combines phased hypothesis exploration, branch context management, and context injection for deep investigation.
- An evaluation on real-world distributed-system bugs that characterizes localization gains, model-dependent behavior, and the computational cost of structured investigation.

2 MOTIVATION

Given an issue report and repository, a diagnosis agent must decide where to look, which explanations the evidence supports, and which code is responsible. Finding related code and selecting the cause pose distinct challenges.

Agents can reach relevant files yet stop at a plausible local explanation without pinpointing the responsible function or source region. Across the examined baseline configurations, file-level localization exceeds source-region localization by up to 28.0 percentage points[1], consistent with the gap reported on a recent benchmark (Anonymous, 2026).

ZooKeeper-2845 illustrates this gap. A former leader retains a transaction in its log that the other replicas never accepted. After a role change, it advertises an older transaction identifier from memory and receives an empty synchronization response, leaving the extra log entry intact. A later restart replays that entry and recreates data absent from the other replicas. The synchronization path explains why the log is not truncated, but leaves a question: why does the server advertise an identifier that disagrees with its own log? Following the database lifecycle leads to `ZooKeeperServer.shutdown`, which retains the stale in-memory state. Several archived agents describe the memory–disk mismatch yet report a getter or loading routine instead of this lifecycle boundary (Appendix A). The diagnostic challenge is to distinguish code that exposes the inconsistency from code that allows it to persist.

An agent's initial search scope may also omit relevant parts of the repository. Across the evaluated mini-SWE-agent configurations, 62.5% of attempts that fail to locate any responsible source region do not inspect a responsible file in their first two code-inspection calls, compared with 28.7% among

[1]These statistics use the configurations and shared tasks in the main evaluation. Appendix G gives the per-configuration results, localization measures, and trace eligibility criteria.

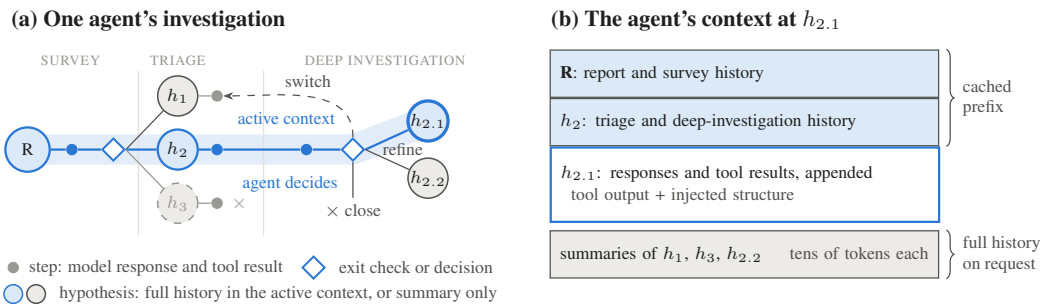


Figure 1: CIRDAN's investigation. (a) One agent surveys at the root R, triages each proposed hypothesis, and investigates a selected one in depth. After a deep investigation, the agent closes, refines, or switches; otherwise the harness takes the highest-scored open leaf. A survey diagnosis ends the run only if it accounts for the report's failure sequence; a deep-investigation diagnosis, once compared with the open hypotheses. (b) The request at $h_{2.1}$ carries full histories along the active context, which form a cached prefix, and summaries of other branches. Exit rules follow the light configuration (§4.1).

attempts that locate all responsible source regions. These pooled rates use traces with unambiguous early inspections. Making related code visible beyond the agent's chosen queries gives it additional directions to investigate. This motivates *context injection*: augmenting tool results with repository relationships that expose code outside the current search scope (§3.3). Appendix G.2 examines subsequent search expansion and variation across attempts.

3 CIRDAN DESIGN

CIRDAN diagnoses a failure from an issue report, the repository at the target revision, and available artifacts, and returns the responsible code locations with a causal explanation. Doing so requires pursuing one explanation in depth while keeping the others open, each with the evidence gathered for it (§2). A single, ever-growing interaction history handles this poorly: reasoning about one hypothesis accumulates alongside stale assumptions from others, and models degrade as long multi-turn histories build up (Laban et al., 2026). The agent's initial search scope can also omit the responsible code.

CIRDAN addresses these challenges with as little structure as possible. The harness holds only what a single agent cannot reliably keep across a long investigation: which explanations remain open, what evidence each has gathered, and when a diagnosis may end the run. The model keeps every other decision. Within each phase it chooses its own tool calls in a ReAct loop (Yao et al., 2023b), and after each investigation it decides whether to close, refine, or leave the current explanation. Leaving these decisions to the model lets it decide where to spend investigation effort, so the harness can benefit as models improve. §4.4 compares this design with multi-agent layouts on the same bugs (Table 2).

Phased hypothesis exploration (Figure 1a) organizes the investigation as a tree that the agent grows and navigates (§3.1). Branch context management determines what the agent sees at each node, its *active context* (Figure 1b; §3.2). Context injection widens the search beyond the agent's own queries (§3.3). The tree, the context layout, and the injection are the same for every model; phase budgets, guidance, and exit rules adapt to how each model reasons (§4.1, §4.3).

3.1 PHASED HYPOTHESIS EXPLORATION

Ranking hypotheses by initial plausibility invites the premature commitment described in §2, while investigating every hypothesis in depth is too expensive. CIRDAN therefore spends effort in three phases of increasing depth: a shared survey, a brief triage pass for each hypothesis, and deep investigation of selected branches.

Investigation state. The investigation is a tree of hypotheses. Its root stores the report and the survey history. Every other node holds a *hypothesis*: an explanation with its investigation plan, a score and its rationale, a status and result, the model and tool history gathered for it, and a summary for other branches. Children refine their parent's explanation, and a *branch* is the path from the root to a node. The branch under investigation forms the agent's active context (§3.2).

Survey. Hypotheses drawn only from the bug report tend to point at the subsystem where the symptoms appear. The survey is therefore a linear investigation: the agent inspects the supplied artifacts and reads code in potentially relevant components, then proposes hypotheses based on what it found. A short budget limits how far it can pursue one explanation before forming alternatives, and the prompt asks for component-level explanations, each with an investigation plan. A diagnosis may end the run at this point if it accounts for the report's symptoms and failure sequence with supporting code evidence.

Triage. Each hypothesis then receives a short triage pass with its own budget for code reads and artifact inspection. Triage gathers preliminary evidence before any branch receives a long investigation: the agent closes hypotheses that the evidence rules out, and the rest stay open. On `Ray-22122`, triage rules out a plausible reporting limit that cannot apply to the reported workload (§4.3; Appendix B). Later passes see summaries of earlier ones, and each pass's full history stays on its node.

Deep investigation. The selected hypothesis then receives a deep investigation with its branch restored as the active context. The diagnosis guidance asks the agent to follow symptoms back to the code that creates, retains, or fails to prevent faulty state, instead of stopping where the error is observed. An investigation ends with a diagnosis or with a decision the agent makes itself. It can *close* the hypothesis as a dead end, *refine* it into narrower child hypotheses, or *switch* by naming another open hypothesis it now considers more promising. If it names none, the harness continues with the highest-scored open leaf. The scores are the model's own assessments, adjusted slightly by depth (Appendix C; Algorithm 1).

The harness therefore never decides when to backtrack. It keeps every alternative and its evidence available, so returning to one is cheap and starts from what is already known. In our workload, triage separates most alternatives, so deep investigations rarely switch branches (Appendix H); the branching ablation in §4.4 compares CIRDAN with a linear investigation on the same bugs.

Diagnosis and stopping. When a deep investigation proposes a diagnosis while other hypotheses remain open, the harness asks the model to compare the diagnosis with those alternatives. The comparison accepts the diagnosis or selects an alternative that may explain an upstream cause for investigation. An optional critique also checks the proposed causal chain against the report (Appendix C). The run ends when a diagnosis is accepted, no open hypotheses remain, or the overall budget is exhausted.

3.2 BRANCH CONTEXT MANAGEMENT

A single agent investigating several hypotheses must decide what to remember. Carrying every branch's history in each request mixes reasoning from different hypotheses and raises the cost of every call. A branch's raw history carries its stale assumptions and dead-end tool output: once the model has read the code behind a refuted explanation, it tends to return to that code even while investigating a different hypothesis. Isolating branches entirely avoids that interference but discards what other branches learned. CIRDAN gives the agent the full history along its active context, from the survey root to the current node, and only summaries of everything else.

Other branches contribute summaries containing their hypotheses, investigation status, and reasoning. A summary is typically tens of tokens, while the investigation behind it can be two orders of magnitude larger. Summaries let the agent use what another branch concluded without inheriting the reasoning and tool output that led there. If a summary contains evidence worth examining, the agent can retrieve that node's full history through a tool.

Requests place the stored histories in root-to-node order, followed by the summaries of other branches and the current phase instruction. The survey and unchanged ancestor histories therefore form a common prefix across requests, and within a branch, new responses and tool results are

appended to the existing history. Providers with prompt caching reuse that prefix: cache hits account for 76–88% of CIRDAN's tokens with the GPT models and 84% with Gemini (§4.2; Figure 5). Summaries bound the context carried between branches, and caching bounds the cost of repeating shared history.

3.3 CONTEXT INJECTION

If the initial survey misses a relevant component, later hypotheses may all remain within the same incomplete scope. CIRDAN augments tool results with related repository information so that the agent can find code outside its chosen search terms and directories (§2; Appendix G.2).

When a shell command accesses source files, the tool wrapper inspects those files and appends detected types, imports, inheritance relationships, and related workspace types to the output. The agent receives this information automatically with the requested snippet. It can follow the referenced types using its normal tools or use them to propose another hypothesis. The injected context supplies leads; the agent decides which to follow.

For ZooKeeper-2845, reading `QuorumPeer.java` inside `server/quorum/` also exposes imports of `ZKDatabase` and `ZooKeeperServer` (Figure 6, Appendix C). The agent can follow these classes into the server state and lifecycle code even if its original search was confined to the quorum directory.

An optional architecture reference can also be supplied at the start of the survey (Appendix C). Appendix J reports exploratory comparisons of context injection and other design components.

Implementation. We implement CIRDAN in Python. The agent uses a shell to inspect source and artifacts in the checked-out revision, and a symbol viewer to list classes and functions with their line ranges. The symbol viewer uses tree-sitter (Brunsfeld et al., 2026) with a Universal Ctags (Universal Ctags Team, 2026) fallback and supports Java, Python, Go, Rust, and C/C++. The shell wrapper extracts and retrieves structural context for these languages (§3.3).

4 EVALUATION

We evaluate whether CIRDAN improves diagnosis quality, how its benefits depend on the model, and how much inference it requires. We examine both *recovering responsible code* and *selecting it precisely*. We then examine how the investigation adapts to each model and test alternative explanations for the gains with small ablations.

4.1 EXPERIMENTAL SETUP

Workload and task. Our main comparison uses the same 34 public bugs from 11 distributed systems for every configuration in Table 1. Each task supplies an issue report, a repository at the target revision, and available issue artifacts. The agent returns responsible files, functions, source-line ranges, and a causal explanation in a structured diagnosis. We score the locations against human-curated root-cause annotations. Appendix E details the workload and scoring, and Appendix K.1 discusses possible prior exposure, and Appendix F threats to validity.

Baselines and configurations. We compare CIRDAN with Codex CLI (Bolin, 2026), mini-SWE-agent (SWE-agent Team, 2026; Yang et al., 2024), OpenCode (Anomaly, 2026), and Claude Code (Anthropic, 2026b), using the model–harness pairings in Table 1. mini-SWE-agent provides a compact agent loop with a linear history. We abbreviate Codex CLI and mini-SWE-agent as Codex and Mini. The models are GPT-5.2-Codex (OpenAI, 2025), GPT-6-Astra (OpenAI, 2026b), Claude Opus 5 (Anthropic, 2026a), DeepSeek-V4-Flash (DeepSeek-AI, 2026), and Gemini-3.8-Flash (Google DeepMind, 2026); we abbreviate the last four as GPT-6, Opus, DeepSeek, and Gemini. Comparisons use the same model and shared bugs, but evaluate complete harness configurations whose prompts, tools, and budgets can differ.

Investigation configurations. We use two CIRDAN configurations. The *guided* configuration, developed with GPT-5.2-Codex, combines a longer initial survey, six-step triage, and detailed guidance on tracing causal origins; deep investigation has no per-branch step limit but remains subject to the

Table 1: Diagnosis quality on the 34 shared bugs (%). Bold marks the best score per model. Block coverage requires line overlap without tolerance. Codex, Mini, Claude, and OC are Codex CLI, mini-SWE-agent, Claude Code, and OpenCode.

Metric		GPT-5.2-Codex				GPT-6-Astra			Opus 5		DS-V4-Flash		Gem-3.8-Flash	
		CIRDAN	Codex	Mini	OC	CIRDAN	Codex	Mini	CIRDAN	Claude	CIRDAN	Mini	CIRDAN	Mini
File	Cov.	**72.5**	56.6	69.2	65.8	80.8	70.3	**88.8**	**85.0**	77.9	61.7	**73.0**	69.5	**69.9**
	Prec.	59.8	**63.2**	51.8	61.8	**77.0**	72.5	61.5	65.1	**71.2**	**58.1**	56.0	**73.5**	72.5
	F1	**65.5**	59.7	59.2	63.7	**78.8**	71.4	72.6	73.7	**74.4**	59.8	**63.4**	**71.5**	71.2
Func	Cov.	**53.0**	32.9	44.0	45.0	58.0	45.3	**68.3**	**71.7**	59.2	42.4	**51.8**	**51.6**	47.8
	Prec.	**43.1**	39.2	30.0	42.6	**55.7**	48.7	38.9	44.2	**44.9**	**37.1**	33.1	**50.7**	48.9
	F1	**47.6**	35.8	35.7	43.8	**56.8**	46.9	49.6	**54.7**	51.1	39.6	**40.4**	**51.2**	48.3
Line	Cov.	**46.3**	26.7	41.8	36.0	51.0	34.5	**60.3**	**66.0**	57.4	32.4	**50.9**	**43.6**	37.4
	Prec.	16.8	**19.2**	11.2	16.2	32.4	**38.4**	15.7	19.3	**23.0**	**14.4**	12.7	25.6	**31.2**
	F1	**24.7**	22.4	17.7	22.3	**39.6**	36.3	24.9	29.9	**32.8**	20.0	**20.3**	32.2	**34.0**
Block	Cov.	**51.5**	28.8	45.2	39.6	58.1	46.1	**60.8**	**69.2**	61.8	37.2	**54.9**	**50.9**	47.4
	Prec.	45.9	44.9	37.2	**47.4**	**61.4**	54.8	42.2	47.7	**52.3**	39.1	**40.2**	**57.4**	51.8
	F1	**48.5**	35.1	40.8	43.2	**59.7**	50.1	49.8	56.5	**56.6**	38.1	**46.4**	**54.0**	49.5
Tokens (M)		1.724	0.614	0.830	0.624	0.432	0.191	0.103	1.334	0.797	1.378	2.309	4.115	10.299
Blocks		2.01	1.49	2.37	1.79	1.81	1.68	3.80	3.21	2.55	1.79	2.99	1.60	1.78

Notes. Scores average attempts within each bug, then bugs; F1 uses aggregate coverage and precision. Tokens (millions) and Blocks (code blocks per diagnosis) are per attempt.

overall run budget. The *light* configuration, developed with GPT-6, uses lighter diagnosis guidance, a smaller survey budget, and stricter acceptance conditions: a survey diagnosis must account for the report's symptoms and failure sequence with supporting code evidence, and triage cannot end the investigation. GPT-5.2-Codex uses the guided configuration; the other models use configurations derived from the light one, and the harness was not tuned for Opus or Gemini. This adaptation is model-specific, as is common for coding agents (MacCallum & Fioca, 2026): budgets, guidance, and acceptance conditions follow how long and how efficiently a model reasons, with no benchmark-specific rules about which code counts as the cause. Neither configuration trains or fine-tunes the model.

Metrics. We report coverage and precision at file, function, line, and block granularity. Coverage measures how much annotated code the diagnosis recovers; precision measures how much reported code matches the annotations. Block coverage requires source-line overlap without a line tolerance, while block precision retains the evaluator's tolerant matching. Line metrics additionally expose whether the reported ranges include substantial surrounding code.

Coverage is our primary metric, since a missed root cause is the costlier error. Coverage alone cannot penalize reporting the cause together with many unrelated locations, so we also report precision, their harmonic mean F1, and the number of blocks per diagnosis. Metrics are averaged within each bug, then equally across bugs; F1 uses unrounded aggregate coverage and precision. Averaging three attempts per bug reduces run-to-run variance; for paired differences between harnesses, we report 95% bootstrap intervals over bugs, which capture variation across the bug sample.

4.2 Comparison with Coding Agents

Accuracy. CIRDAN achieves the highest block coverage with GPT-5.2-Codex, Opus, and Gemini, and matches mini-SWE-agent with GPT-6 while reporting half as many blocks (Table 1 and fig. 2). With GPT-5.2-Codex, CIRDAN recovers 51.5% of annotated blocks, against 28.8–45.2% for the three baselines; its gains over Codex (22.7 points) and OpenCode (11.8 points) have 95% intervals that exclude zero. With Opus and Gemini, for which the harness was not tuned, CIRDAN raises coverage from 61.8% to 69.2% and from 47.4% to 50.9%. With GPT-6, CIRDAN covers 12.0 points more than Codex. Mini's slightly higher coverage (60.8% versus 58.1%) comes with 3.8 blocks per diagnosis, twice as many as CIRDAN: it reports much of the code it has read instead of selecting the

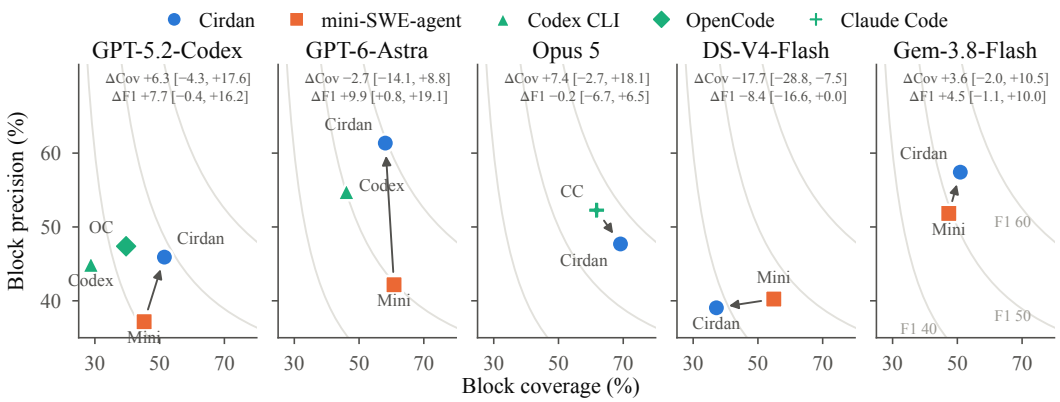
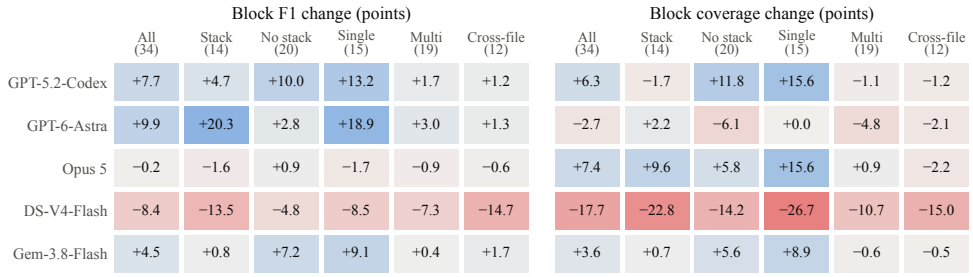


Figure 2: Block coverage and precision per model on the 34 shared bugs. Arrows run from the baseline (mini-SWE-agent; Claude Code for Opus) to CIRDAN; labels give the change in coverage and F1, in points, with 95% bootstrap intervals over bugs. Gray curves mark constant F1.



Block F1 change (points)

	All (34)	Stack (14)	No stack (20)	Single (15)	Multi (19)	Cross-file (12)
GPT-5.2-Codex	+7.7	+4.7	+10.0	+13.2	+1.7	+1.2
GPT-6-Astra	+9.9	+20.3	+2.8	+18.9	+3.0	+1.3
Opus 5	−0.2	−1.6	+0.9	−1.7	−0.9	−0.6
DS-V4-Flash	−8.4	−13.5	−4.8	−8.5	−7.3	−14.7
Gem-3.8-Flash	+4.5	+0.8	+7.2	+9.1	+0.4	+1.7

Block coverage change (points)

	All (34)	Stack (14)	No stack (20)	Single (15)	Multi (19)	Cross-file (12)
GPT-5.2-Codex	+6.3	−1.7	+11.8	+15.6	−1.1	−1.2
GPT-6-Astra	−2.7	+2.2	−6.1	+0.0	−4.8	−2.1
Opus 5	+7.4	+9.6	+5.8	+15.6	+0.9	−2.2
DS-V4-Flash	−17.7	−22.8	−14.2	−26.7	−10.7	−15.0
Gem-3.8-Flash	+3.6	+0.7	+5.6	+8.9	−0.6	−0.5

Figure 3: CIRDAN's change over the baseline (mini-SWE-agent; Claude Code for Opus) in block F1 (left) and block coverage (right), in points, by report evidence and root-cause structure. Blue marks gains and red losses; bug counts are in parentheses.

cause, and its precision is 42.2% against 61.4% for CIRDAN. DeepSeek is the exception, with Mini ahead on both coverage and precision (§4.3).

Where structure helps. Coverage gains concentrate on bugs whose cause is a single annotated block (Figure 3): CIRDAN raises block coverage there by 15.6 points with GPT-5.2-Codex and Opus and by 8.9 with Gemini, while gains on multi-block and cross-file bugs stay within one point. CIRDAN's diagnoses name one or two blocks, so causes spread across several locations benefit less. GPT-5.2-Codex and Gemini also gain most on reports without a stack trace (11.8 and 5.6 points). With GPT-6, the gain is in selection and is largest when the report includes a stack trace, where block F1 rises by 20.3 points: on these bugs, mini-SWE-agent reports 4.2 blocks per diagnosis, twice as many as CIRDAN, and its precision is 42% against 74%.

Cost. CIRDAN costs more, mostly as context rather than model calls, and most of that context is served from cache (Figures 4 and 5). It uses 2.1–2.8× the tokens of the GPT-5.2-Codex baselines, 2.3–4.2× those of the GPT-6 baselines, and 1.7× those of Claude Code, while making 1.3–1.6× the model calls of mini-SWE-agent and about as many as Claude Code: each request carries the survey history and summaries of other branches. Cache hits account for 76–88% of CIRDAN's tokens with the GPT models and 84% with Gemini, so where cached input is inexpensive, as in local serving, the effective cost is well below the totals. Reuse is lower with Opus (57%), which emits about four times more output per call, all of which must first be written to the cache. With Gemini, CIRDAN uses 2.5× fewer tokens than Mini.

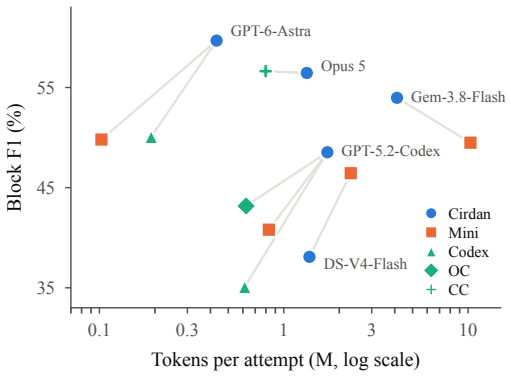


Figure 4: Block F1 against tokens per attempt (log scale). Lines connect each baseline to CIRDAN on the same model.

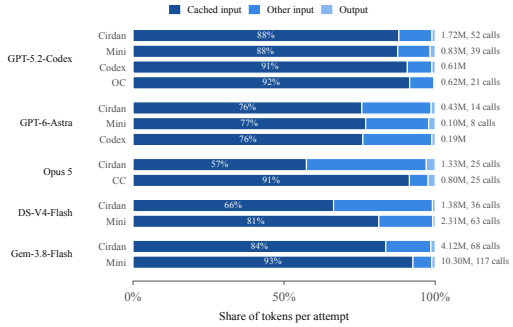


Figure 5: Tokens per attempt by type (cached input, other input, output), with totals and model calls; Codex CLI reports one aggregate call.

4.3 MATCHING STRUCTURE TO THE MODEL

Models fail in different ways, and structure helps when it targets the failure a model actually has. A missed root cause has two sources. In a *discovery* failure, the agent never inspects the responsible code during the investigation. In a *selection* failure, it inspects that code but leaves it out of the diagnosis, or reports it among unrelated locations. GPT-5.2-Codex mainly needs help with discovery: CIRDAN raises its block coverage over every baseline, by 6.3 to 22.7 points, at precision close to the most precise baseline. GPT-6 discovers relevant code on its own, since Mini already covers slightly more than CIRDAN, and its gain comes from selection.

The two models also investigate differently. With GPT-5.2-Codex, a CIRDAN attempt issues 48 tool calls, nearly one shell command each, and no attempt ends in the survey. GPT-6 issues 10 calls with about five commands in each, and 73% of its attempts end in the survey: it reaches a plausible diagnosis early and with confidence. The two configurations follow these patterns (§4.1). The guided configuration gives GPT-5.2-Codex a longer investigation that pushes it toward the relevant code. The light configuration gives GPT-6 a smaller survey budget but stricter acceptance, so an early diagnosis must account for the report's full failure sequence before the run ends. On `Ray-22122` with GPT-5.2-Codex, triage rules out a plausible reporting limit before deep investigation; on `HBase-28376` with GPT-6, further investigation moves the attribution from startup code to the incomplete migration step (Appendix B).

The same structure can also get in the way. DeepSeek reasons at length within each call, and budgets derived for GPT-6's short investigations cut its exploration short: under CIRDAN it runs 53 shell commands per attempt against Mini's 139, uses 1.7× fewer tokens, and ends 38% of its attempts in the survey. Recovering its accuracy requires budgets and guidance fitted to its long reasoning, a non-trivial adaptation we leave to future work.

4.4 ABLATIONS

Three small ablations with GPT-6 test alternative explanations for these gains (Table 2). More reasoning does not close the gap: on five bugs, raising Codex's effort from medium to xhigh moves block F1 from 44.9% to 45.5% at 1.5× the cost, whereas CIRDAN at medium effort reaches 68.6%. Branching matters: on the same bugs, CIRDAN with branching reaches 73.6% block F1 against 54.8% for a linear CIRDAN at xhigh, at 2.84× the known cost (two linear attempts timed out and score zero). Communicating agents do no better than one: on eight bugs, native delegation, parallel hypothesis investigators, and role decomposition change block F1 by +1.3, −1.2, and −13.8 points relative to a single investigator, while CIRDAN reaches the highest F1, 87.5%, at 2.93× the single investigator's tokens. Appendices I and H detail the first two ablations.

Table 2: Ablations with GPT-6-Astra. Effort and branching: five bugs, two attempts each. Multi-agent: eight bugs, one max-effort attempt in a source-only harness; rows below the rule are main-table runs, and coverage uses a five-line tolerance. Compute is relative to each block's first row.

Study	Configuration	Block cov.	Block prec.	Block F1	Compute
Reasoning effort (5 bugs)	Codex, medium effort	46.7	43.3	44.9	1.00×
	Codex, xhigh effort	50.0	41.7	45.5	1.50×
	CIRDAN, medium effort	**65.8**	**71.7**	**68.6**	1.23×
Branching (5 bugs)	CIRDAN without branching, xhigh	58.3	51.7	54.8	1.00×
	CIRDAN with branching, medium	**70.8**	**76.7**	**73.6**	2.84×
Multi-agent (8 bugs)	Single investigator	93.8	70.8	80.7	1.00×
	Native delegation	93.8	72.9	82.0	1.63×
	Hypothesis investigators	87.5	72.9	79.5	1.85×
	Role decomposition	75.0	60.4	66.9	0.84×
	Codex, medium	66.7	72.2	69.3	1.20×
	mini-SWE-agent, medium	**95.8**	50.1	65.8	0.58×
	CIRDAN, medium	89.6	**85.4**	**87.5**	2.93×

5 Related Work

Coding-agent harnesses and search. ReAct interleaves reasoning and tool use (Yao et al., 2023b), a pattern used by coding agents such as SWE-agent and OpenHands (Yang et al., 2024; Wang et al., 2025). Agentless uses a fixed workflow for repository-level patch generation (Xia et al., 2025), while Tree of Thoughts searches alternative reasoning paths (Yao et al., 2023a). LATS and SWE-Search apply tree search to agent trajectories, expanding promising actions using value estimates or execution feedback (Zhou et al., 2024; Antoniades et al., 2025). CIRDAN specializes structured search for source-level diagnosis, where evidence may be incomplete and a reliable reproducer unavailable. Its branches retain causal hypotheses and code evidence, while phased exploration and stopping rules determine how much investigation each explanation receives.

Multi-agent systems. Systems such as ChatDev assign software-development roles to communicating agents (Qian et al., 2024). Triangle routes incidents through analyzer, decider, and per-team agents that negotiate by vote (Yu et al., 2025a), and the OpenRCA benchmark's RCA-agent pairs a controller agent with a code-executing agent (Xu et al., 2025). CIRDAN uses one agent across multiple hypothesis states. Each keeps its history, and summaries share findings across branches.

Diagnosis and program analysis. Diagnosis solutions localize faults from executions, logs, and traces (Zeller, 2002; Yuan et al., 2010; Zhang et al., 2019; Fonseca et al., 2007; Quinn et al., 2022). Recent LLM-based work targets incident management: OpenRCA benchmarks root-cause identification over logs, metrics, and traces (Xu et al., 2025), Xpert generates telemetry queries (Jiang et al., 2024), and Cloud Atlas builds causal graphs for fault localization (Xie et al., 2024). DistDebug-Bench evaluates agents on source-level diagnosis of distributed-system failures (Anonymous, 2026); we use a subset of its tasks. Issue-localization agents use code search or dependency graphs to find the code an issue concerns, usually as a step toward a patch (Zhang et al., 2024; Qin et al., 2025; Yu et al., 2025b; Chen et al., 2025). CIRDAN's context injection exposes similar structural relationships inside ordinary shell output, and CIRDAN targets the root cause of cross-component failures rather than the location of a requested change.

6 Conclusion

We presented CIRDAN, a single-agent harness for source-level diagnosis of distributed-system failures that combines phased hypothesis exploration, branch context management, and context injection. On 34 bugs, it improves block coverage or precision with four of five models at up to 4.2× the baselines' tokens, most served from cache. Its budgets and guidance must match how each model reasons, and long-reasoning models such as DeepSeek need further adaptation; training model and harness together could improve this fit (MacCallum & Fioca, 2026).

AI USE STATEMENT

We used generative AI tools primarily to assist with running experiments and drafting and revising the manuscript, with additional assistance in planning experimental comparisons and developing supporting scripts. The core research implementation, including the CIRDAN harness and experiment-running code, was substantially written by the authors to maintain control over experimental rigor and correctness. The authors take responsibility for the experimental procedures, interpretation of results, and final content of the paper, including all AI-assisted work.

ETHICS STATEMENT

This work studies diagnosis using publicly available software repositories and bug reports. CIRDAN may produce incorrect or incomplete causal explanations; its outputs should be treated as diagnostic suggestions and independently validated before applying code changes or operational interventions. Our evaluation measures source-level localization and does not establish the safety or correctness of automated repairs.

REPRODUCIBILITY STATEMENT

We will open-source all implementation and evaluation code and provide a one-command script to rerun the experiments and regenerate the reported tables and figures after installing the documented dependencies and configuring model access. The release will include the prompts, configurations, benchmark revision identifiers, and evaluation protocol. §3 describes the method; §4.1 and Appendix E specify the main evaluation; Appendix C describes the runtime and prompts. Reproduction should match the model provider and version, inference settings, execution budgets, and cache configuration; cost comparisons also require matching pricing assumptions. Stochastic variation and unannounced changes to provider-side models or inference services can affect outcomes even when exposed settings are unchanged. The workflow will support repeated-run comparison of aggregate results rather than guarantee identical trajectories or scores.

REFERENCES

Anomaly. OpenCode: The open source coding agent. https://github.com/anomalyco/opencode, 2026. GitHub repository, accessed September 26, 2026.

Anonymous. DistDebug-Bench: Can LLM agents diagnose bugs in distributed systems? In *Advances in Neural Information Processing Systems, Datasets and Benchmarks Track*, 2026.

Anthropic. System card: Claude Opus 5. https://www.anthropic.com/claude-opus-5-system-card, July 2026a. Published July 24, 2026; updated August 19, 2026.

Anthropic. How Claude Code works. Claude Code documentation, 2026b. URL https://code.claude.com/docs/en/how-claude-code-works. Accessed September 24, 2026.

Antonis Antoniades, Albert Örwall, Kexun Zhang, Yuxi Xie, Anirudh Goyal, and William Wang. SWE-Search: Enhancing software agents with Monte Carlo tree search and iterative refinement. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=G7sIFXugTX.

Apache ZooKeeper. ZOOKEEPER-2845: Data inconsistency issue due to retain database in leader election. https://issues.apache.org/jira/browse/ZOOKEEPER-2845, 2017.

Michael Bolin. Unrolling the Codex agent loop. OpenAI Engineering, January 2026. URL https://openai.com/index/unrolling-the-codex-agent-loop/. Published January 23, 2026.

Max Brunsfeld et al. Tree-sitter: A parser generator tool and an incremental parsing library. https://github.com/tree-sitter/tree-sitter, 2026. Accessed September 26, 2026.

Zhaoling Chen, Xiangru Tang, Gangda Deng, Fang Wu, Jialong Wu, Zhiwei Jiang, Viktor Prasanna, Arman Cohan, and Xingyao Wang. LocAgent: Graph-guided LLM agents for code localization. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 8697–8727, 2025. doi: 10.18653/v1/2025.acl-long.426.

DeepSeek-AI. DeepSeek-V4: Towards highly efficient million-token context intelligence, 2026. URL https://arxiv.org/abs/2606.19348.

Rodrigo Fonseca, George Porter, Randy H. Katz, Scott Shenker, and Ion Stoica. X-Trace: A pervasive network tracing framework. In *4th USENIX Symposium on Networked Systems Design & Implementation (NSDI 07)*, pp. 271–284, Cambridge, MA, April 2007. USENIX Association. URL https://www.usenix.org/conference/nsdi-07/x-trace-pervasive-network-tracing-framework.

Google DeepMind. Gemini 3.8 Flash model card. https://storage.googleapis.com/deepmind-media/Model-Cards/Gemini-3-8-Flash-Model-Card.pdf, September 2026. Published September 2, 2026.

Yuxuan Jiang, Chaoyun Zhang, Shilin He, Zhihao Yang, Minghua Ma, Si Qin, Yu Kang, Yingnong Dang, Saravan Rajmohan, Qingwei Lin, and Dongmei Zhang. Xpert: Empowering incident management with query recommendations via large language models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, pp. 1–13, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400702174. doi: 10.1145/3597503.3639081. URL https://doi.org/10.1145/3597503.3639081.

Philippe Laban, Hiroaki Hayashi, Yingbo Zhou, and Jennifer Neville. LLMs get lost in multi-turn conversation. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=VKGTGGcwl6.

Noah MacCallum and Brian Fioca. Codex prompting guide. OpenAI Cookbook, February 2026. URL https://developers.openai.com/cookbook/examples/gpt-5/codex_prompting_guide. Accessed September 25, 2026.

Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. Pivot tracing: Dynamic causal monitoring for distributed systems. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pp. 378–393, New York, NY, USA, 2015. Association for Computing Machinery. doi: 10.1145/2815400.2815415. URL https://doi.org/10.1145/2815400.2815415.

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

OpenAI. Addendum to GPT-5.2 system card: GPT-5.2-Codex. https://cdn.openai.com/pdf/ac7c37ae-7f4c-4442-b741-2eabdeaf77e0/oai_5_2_Codex.pdf, December 2025. Published December 18, 2025.

OpenAI. GPT-5.3-Codex system card. https://cdn.openai.com/pdf/23eca107-a9b1-4d2c-b156-7deb4fbc697c/GPT-5-3-Codex-System-Card-02.pdf, February 2026a. Published February 5, 2026.

OpenAI. GPT-6 Astra system card. https://deploymentsafety.openai.com/gpt-6-astra/, September 2026b. Published September 3, 2026; updated September 22, 2026.

Jia Pan, Haoze Wu, Tanakorn Leesatapornwongsa, Suman Nath, and Peng Huang. Efficient reproduction of fault-induced failures in distributed systems with feedback-driven fault injection. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, SOSP '24, pp. 46–62, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400712517. doi: 10.1145/3694715.3695979. URL https://doi.org/10.1145/3694715.3695979.

Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, Juyuan Xu, Dahai Li, Zhiyuan Liu, and Maosong Sun. ChatDev: Communicative agents for software development. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 15174–15186, 2024. doi: 10.18653/v1/2024.acl-long.810. URL https://aclanthology.org/2024.acl-long.810/.

Yihao Qin, Shangwen Wang, Yiling Lou, Jinhao Dong, Kaixin Wang, Xiaoling Li, and Xiaoguang Mao. SoapFL: A standard operating procedure for LLM-based method-level fault localization. *IEEE Transactions on Software Engineering*, 51(4):1173–1187, 2025. doi: 10.1109/TSE.2025.3543187.

Andi Quinn, Jason Flinn, Michael Cafarella, and Baris Kasikci. Debugging the OmniTable way. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pp. 357–373, Carlsbad, CA, July 2022. USENIX Association. ISBN 978-1-939133-28-1. URL https://www.usenix.org/conference/osdi22/presentation/quinn.

SWE-agent Team. mini-SWE-agent documentation, 2026. URL https://mini-swe-agent.com/latest/. Accessed September 25, 2026.

Universal Ctags Team. Universal Ctags: A maintained implementation of ctags. https://github.com/universal-ctags/ctags, 2026. Accessed September 26, 2026.

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, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. OpenHands: An open platform for AI software developers as generalist agents. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=OJd3ayDDoF.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Demystifying LLM-based software engineering agents. *Proc. ACM Softw. Eng.*, 2(FSE):801–824, July 2025. doi: 10.1145/3715754. URL https://doi.org/10.1145/3715754.

Zhiqiang Xie, Yujia Zheng, Lizi Ottens, Kun Zhang, Christos Kozyrakis, and Jonathan Mace. Cloud Atlas: Efficient fault localization for cloud systems using language models and causal insight, 2024. URL https://arxiv.org/abs/2407.08694.

Junjielong Xu, Qinan Zhang, Zhiqing Zhong, Shilin He, Chaoyun Zhang, Qingwei Lin, Dan Pei, Pinjia He, Dongmei Zhang, and Qi Zhang. OpenRCA: Can large language models locate the root cause of software failures? In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=M4qNIzQYpd.

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik R Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems*, volume 37, pp. 50528–50652, 2024. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/5a7c947568c1b1328ccc5230172e1e7c-Abstract-Conference.html.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. In *Advances in Neural Information Processing Systems*, volume 36, pp. 11809–11822, 2023a. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/271db9922b8d1f4dd7aaef84ed5ac703-Abstract-Conference.html.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations*, 2023b. URL https://openreview.net/forum?id=WE_vluYUL-X.

Zhaoyang Yu, Aoyang Fang, Minghua Ma, Jaskaran Singh Walia, Chaoyun Zhang, Shu Chi, Ze Li, Murali Chintalapati, Xuchao Zhang, Rujia Wang, Chetan Bansal, Saravan Rajmohan, Qingwei Lin, Shenglin Zhang, Dan Pei, and Pinjia He. Triangle: Empowering incident triage with multi-agent. In *Proceedings of the 40th IEEE/ACM International Conference on Automated Software Engineering*, pp. 674–686, 2025a. doi: 10.1109/ASE63991.2025.00062. URL https://doi.org/10.1109/ASE63991.2025.00062.

Zhongming Yu, Hejia Zhang, Yujie Zhao, Hanxian Huang, Matrix Yao, Ke Ding, and Jishen Zhao. OrcaLoca: An LLM agent framework for software issue localization. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 73416–73436, 2025b. URL https://proceedings.mlr.press/v267/yu25x.html.

Ding Yuan, Haohui Mai, Weiwei Xiong, Lin Tan, Yuanyuan Zhou, and Shankar Pasupathy. SherLog: Error diagnosis by connecting clues from run-time logs. In *Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pp. 143–154, New York, NY, USA, March 2010. Association for Computing Machinery. ISBN 9781605588391. doi: 10.1145/1736020.1736038. URL https://doi.org/10.1145/1736020.1736038.

Andreas Zeller. Isolating cause-effect chains from computer programs. In *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering*, SIGSOFT '02/FSE-10, pp. 1–10, New York, NY, USA, 2002. Association for Computing Machinery. ISBN 1581135149. doi: 10.1145/587051.587053. URL https://doi.org/10.1145/587051.587053.

Yongle Zhang, Kirk Rodrigues, Yu Luo, Michael Stumm, and Ding Yuan. The inflection point hypothesis: a principled debugging approach for locating the root cause of a failure. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pp. 131–146, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450368735. doi: 10.1145/3341301.3359650. URL https://doi.org/10.1145/3341301.3359650.

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

Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. Language agent tree search unifies reasoning, acting, and planning in language models. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 62138–62160, 2024. URL https://proceedings.mlr.press/v235/zhou24r.html.

A ZooKeeper-2845 Case Study

This case connects the observations in §2 to concrete investigation behavior. We describe the task and the recorded outcomes, show how investigations follow report anchors and diverge in their search paths, and trace how CIRDAN organizes the same failure as competing hypotheses.

A.1 Task and Recorded Outcomes

ZooKeeper-2845 begins with three replicas at committed transaction T_0: A and B are followers and C is the leader (Apache ZooKeeper, 2017). C writes a new transaction T_1 to its local log, but A and B restart before receiving the proposal. A and B form a new quorum whose state ends at T_0. When C changes role and synchronizes with the new leader, its retained in-memory database still reports T_0 even though its log contains T_1. An empty `DIFF` leaves the extra log entry intact. C then restarts before taking a new snapshot, replays T_1, and recreates data absent from A and B. The reference diagnosis locates the retained-state behavior in `ZooKeeperServer.shutdown`, lines 555–558.

Table 3 summarizes three submitted diagnoses for each of ten Mini-SWE-agent model configurations, drawn from separate evaluations. Seven configurations miss the annotated block at least once; three miss it in all three attempts. Overall, 14 of 30 attempts miss the block. The table reports localization; an attempt that misses the block can still explain part of the failure correctly.

A.2 Report Anchors and Untested Alternatives

The report emphasizes quorum formation and synchronization, making `server/quorum/` a natural starting point. That subsystem explains how an advertised zxid leads to an empty `DIFF`. A further question is why the former leader advertises T_0 when its durable log contains T_1. Answering it requires following the lifecycle of the retained database, whose shutdown code lies in `ZooKeeperServer.java`, directly under `server/`.

Two of the three GPT-5.2-Codex diagnoses name the correct file but miss its annotated shutdown block. In one of them, the agent reads the file's database-loading code and reports

Model (Mini-SWE-agent)	Attempts missing the annotated block
GPT-5.2-Codex	3/3
GPT-5.3-Codex	1/3
GPT-5.4	0/3
GPT-5.5	2/3
DeepSeek-V4-Flash	1/3
Kimi-K2.6	0/3
Gemini-3.5-Flash	0/3
Hunyuan3-Preview	3/3
MiMo-V2.5	3/3
MiMo-V2.5-Pro	1/3

Table 3: ZooKeeper-2845 localization outcomes in ten archived Mini-SWE-agent model configurations. Seven models miss the annotated shutdown block in at least one attempt; three miss it in every attempt. All 30 attempts produced submitted diagnoses.

`ZooKeeperServer.loadData` together with `QuorumPeer.getLastLoggedZxid`. Its explanation already recognizes the mismatch between retained memory and the durable log, the empty `DIFF`, and replay after restart, but it never identifies the shutdown block that retains the database across the transition. The agent has a largely correct account of the failure and scores zero block coverage because it did not investigate the lifecycle path. The review criteria for counting such premature commitment appear in Appendix G.3.

A.3 Search Paths and Final Code Attribution

Two GPT-5.3-Codex attempts both begin with queries about database retention and the report's reference to an earlier issue, then diverge in their source reads. In the first, the agent examines loading, zxid reporting, role changes, and synchronization. It reads `ZooKeeperServer.java`, but its final diagnosis names `QuorumPeer.getLastLoggedZxid`, `QuorumPeer.makeFollower`, and `Learner.syncWithLeader`, missing the annotated block.

In the second, the agent also searches for database clearing and follows `shutdown` into `ZooKeeperServer.java`. After reading lines 500–560 and revisiting the shutdown region with line numbers, it reports `ZooKeeperServer.shutdown` together with downstream zxid reporting and leader-side synchronization. This attempt covers the annotated block.

Both final explanations describe retained database state, a stale advertised zxid, and replay of an extra durable transaction. They differ in which paths they inspect and hold responsible: similar initial queries lead to different code attribution.

A.4 Connection to Structured Hypothesis Exploration

The failure supports three investigative directions: *quorum synchronization*, asking whether `DIFF` selection or truncation is incorrect; *persistence*, asking why restart replays an uncommitted transaction; and *lifecycle*, asking how retained database state becomes inconsistent with the log. In the representative CIRDAN trace, four hypothesis nodes cover these directions: two examine database retention, one examines the `DIFF` path, and one examines startup and zxid reporting.

The final diagnosis connects retained database state during shutdown to the stale zxid, skipped log truncation, and replay after restart. It reports the annotated `ZooKeeperServer.shutdown` block and also names `QuorumPeer.getLastLoggedZxid` as a downstream mechanism that explains propagation. Keeping the lifecycle hypothesis open lets the investigation return to it while it examines the synchronization path.

The trace illustrates the mechanism; the workload-level comparisons appear in Appendix J.

B Investigation Trajectories

The final code locations hide the decisions that produced them. An agent may need to choose between explanations in different subsystems, or it may already understand the failure but still need to decide which part of the code is responsible. The following trajectories illustrate both decisions.

Testing a plausible explanation. In `Ray-22122`, the autoscaler's resource-demand report understates queued work. A CIRDAN investigation with GPT-5.2-Codex considers three places where the information could be lost: the worker counting its backlog, the raylet reporting resource demand, and the autoscaler aggregating those reports. The raylet contains a reporting limit, making it a plausible suspect. Triage reveals that this limit caps the number of distinct scheduling classes; the reported workload has only a single CPU shape. That constraint rules out the proposed truncation mechanism. The investigation then concentrates on worker-side backlog accounting and reports the responsible code there. Here, inspecting an alternative produces a concrete reason to set it aside before committing to the deeper investigation (Appendix J).

Choosing where to attribute the failure. On `HBase-28376`, CIRDAN recognizes during survey that migration writes into a missing destination column family. Its early proposed diagnosis names startup code and a descriptor helper. Further investigation distinguishes fresh initialization from an upgrade: creating a new table supplies the family, while loading an existing descriptor preserves the old schema. The final answer points to the migration routine, which writes into the new family without ensuring that it exists. The broad explanation survives, but the code attribution moves from startup and descriptor handling to the incomplete migration step. This trace comes from the branching ablation in §4.4; Appendix H gives its per-bug results.

C CIRDAN Runtime and Prompts

This section describes the CIRDAN runtime: its state, control loop, scoring, context construction, and prompts. The defaults below use 15 survey steps, 8 triage steps, and optional critique. The guided and light configurations of the main evaluation change these budgets, prompts, and acceptance rules (§4.1); §4 describes the evaluated configurations, and Appendix L how the light configuration adapts them to GPT-6.

C.1 Harness State

Each thought node stores a hypothesis and investigation plan, parent and child identifiers, depth, priority score and its rationale, model and tool history, completed phases, reasoning summaries, and the latest investigation result. Comparisons with alternative hypotheses are retained in a verification history. The root contains the original task and survey history; subsequent nodes retain the interactions produced by their own investigations.

The model reports one of three investigation states:

- `diagnosis_provided`: a proposed diagnosis, with the root-cause code locations and a causal explanation;
- `need_more_investigation`: an unresolved explanation with a rationale for continuing;
- `no_more_investigation`: a branch judged unproductive.

The diagnosis prompt requests one or two root-cause code blocks. The evidence and reasoning behind the reported state are stored in the interaction history and rationale. The model proposes hypotheses as a list of explanations and investigation plans. The harness does not impose a fixed branching factor or deduplicate the list.

C.2 Budgets and Stopping Policy

Table 4 lists the default settings used by the harness in Algorithm 1.

Table 4: Default control policy of the CIRDAN runtime.

Component	Default behavior
Survey	15 model iterations, followed by at most two wind-down iterations requesting structured output.
Triage	8 iterations per initial hypothesis with the same wind-down policy; branches returning `no_more_investigation` are pruned.
Deep investigation	No per-node iteration limit. After a non-diagnosing investigation, child hypotheses are proposed and the remaining leaf nodes are rescored.
Global stopping	1,800-second wall-time budget, checked between model calls; terminate on an accepted diagnosis, no leaf nodes left to investigate, or timeout.
Context	Full raw history on the selected root-to-node path; structured summaries for off-path nodes; full sibling traces available on demand.
Diagnosis checks	Optional causal-chain critique; forced comparison with open alternatives only for a deep-stage diagnosis when other leaf nodes remain to be investigated.
Tools	Workspace-rooted shell and symbol viewer; structured stages also receive `get_node_details` and, when enabled, the critique tool.

Survey and triage have iteration limits, after which the agent is asked to conclude within at most two wind-down turns. Deep investigation has no default per-node iteration limit: control returns when the model finishes the branch or the overall time budget is reached. A single deep investigation can use the remaining time budget. The timeout is checked between calls and does not interrupt a model or tool call already in progress.

Under the default policy, survey and triage diagnoses exit directly. A deep-stage diagnosis is compared with the hypotheses in leaf nodes still awaiting investigation. A separate model call judges whether another hypothesis could contain an upstream cause and can recommend a branch to investigate. If it rejects the diagnosis without recommending a node, the harness selects the highest-scored remaining node. A deep diagnosis with no open alternatives exits directly.

The model can invoke the optional critique tool with a proposed causal chain and diagnosis summary. A separate call decomposes the bug report into causal steps and checks whether the proposed explanation supplies concrete file-and-line evidence for the code responsible for each step. The feedback is returned to the investigating agent; the harness neither requires the tool call nor mechanically enforces its verdict.

If a deep investigation returns without a diagnosis, the harness proposes children and rescores all remaining leaf nodes. Unlike triage pruning, this expansion is applied to non-diagnosing deep-stage results regardless of their reported state. If time runs out or no leaf nodes remain to investigate before a diagnosis is accepted, the runtime records failure and retains the investigation artifacts without synthesizing a fallback diagnosis.

In the light configuration, a deep investigation that ends without a diagnosis closes its branch instead: the agent may name another open hypothesis to investigate next, and otherwise the harness selects the highest-scored open leaf. Once no open hypotheses remain, the agent reviews the tree and may add a new hypothesis or stop.

Here, O contains the leaf nodes awaiting investigation. Select uses the node recommended by the preceding comparison when available, and otherwise chooses the highest-scored leaf node. After expansion, scoring uses the latest branch's investigation history and the hypotheses in all remaining leaf nodes.

C.3 SCORING LEAF NODES

The model assigns each open hypothesis a score in $[0, 1]$ and a rationale. Scoring guidance favors broad component explanations near the root, more precise locations deeper in the investigation, and

Algorithm 1: CIRDAN harness. Investigation calls share the global wall-time guard; checks within each call are omitted for clarity.

Input: bug report b, repository R, optional architecture reference a
Output: diagnosis d or failure
$r_0 \leftarrow \text{Survey}(b, R, a; 15)$;
if r_0 *contains a diagnosis* **then**
  return r_0
$O \leftarrow \text{Propose}(r_0.\text{history})$;
foreach $h \in O$ **do**
  $r_h \leftarrow \text{Triage}(h; 8)$;
  if r_h *contains a diagnosis* **then**
    return r_h
  if $r_h.status =$ `no_more_investigation` **then**
    remove h from O
ScoreLeaves(O);
while $O \neq \emptyset$ *and elapsed time* < 1800 *s* **do**
  $h \leftarrow \text{Select}(O)$; remove h from O;
  $r_h \leftarrow \text{DeepInvestigate}(h)$;
  if r_h *contains a diagnosis* **then**
    if $O = \emptyset$ **then**
      return r_h
    $v \leftarrow \text{CompareAlternatives}(r_h, O)$;
    if v *accepts* **then**
      return r_h
    prioritize v's recommended node for the next selection, if present;
    continue;
  $O \leftarrow O \cup \text{ProposeChildren}(h.\text{history})$;
  ScoreLeaves(O);
return *failure*;

code that creates or fails to clear faulty state. It also asks the model to consider alternatives to an unproductive deep investigation.

The runtime applies fixed adjustments to this score. It treats a hypothesis as line-specific when its lowercased text contains a digit together with `line`, `.java`, `.py`, or `.cpp`. Such a node receives -0.15 at depth at most one and $+0.10$ at greater depth. A node receives another $+0.10$ when its text contains any of {`state`, `cleanup`, `init`, `persist`, `write`, `sync`, `flush`, `commit`, `retain`, `clear`, `shutdown`}. The final score is clipped to $[0, 1]$. The resulting scores are heuristic priorities.

C.4 Context Construction and Tool Augmentation

To construct a request, the harness traverses the selected root-to-node path in order. It appends each node's stored interactions followed by its current metadata, then appends summaries of all off-path nodes and the phase instruction. Off-path summaries contain the hypothesis, score, investigation result, completed phases, and reasoning summaries. The `get_node_details` tool returns a requested node's complete stored record, including its model and tool history.

The root's stored interactions form a shared prefix across branches. Further prefix reuse depends on ancestor histories and metadata remaining unchanged; updates to metadata or off-path summaries can change the suffix of a later request. Cache eligibility, retention, and savings depend on the provider.

The shell wrapper detects source paths in commands and extracts structural information from the corresponding files, falling back to command output when file contents are unavailable. It parses the accessed source files and searches the repository for related types. It appends detected types and inheritance information, up to 15 project imports, and up to five related workspace files per detected type. These relationships supply retrieval leads without whole-program call-graph or dataflow analysis. Figure 6 shows the injected context when the agent reads a quorum file for ZooKeeper-2845.

```
$ sed -n '926,931p' src/java/main/org/apache/zookeeper/server/quorum/QuorumPeer.java
926  public long getLastLoggedZxid() {
927      if (!zkDb.isInitialized()) {
928          loadDataBase();
929      }
930      return zkDb.getDataTreeLastProcessedZxid();
931  }
[ENVIRONMENT INJECTED CONTEXT]
- Related imports in file: FileTxnSnapLog, ZKDatabase, ZooKeeperServer, ...
- [Java] QuorumPeer (extends ZooKeeperThread)
- Nearby workspace types/subclasses: QuorumServer, ResponderThread, ...
```

Figure 6: Context injection for ZooKeeper-2845 (abbreviated). Reading a quorum file also exposes `ZKDatabase` and `ZooKeeperServer`, which the agent can follow outside the quorum directory.

```
## Phase 1: Broad Survey
**You have approximately {phase_1_max_steps} tool calls for this phase.**
Your goals are:
1. **Orient**: Read the bug report and artifacts.
2. **Survey the codebase**: Identify key files, classes, and subsystems.
3. **Form hypotheses**: Identify 2--4 plausible root-cause hypotheses.
4. **Conclude**: Summarize findings and list hypotheses for later phases.
5. **High-level Summaries**: Keep findings at the component or conceptual level.
**Do NOT attempt a complete diagnosis in Phase 1** unless the root cause is
immediately obvious.
```

(a) Broad survey prompt

```
## Phase 2: Triage
Use the structured-hypothesis scaffold to briefly investigate each seeded
hypothesis. After each short investigation:
1. record current likelihood and evidence,
2. summarize the branch concisely,
3. leave deeper verification for the next phase.
```

(b) Triage prompt

```
## Phase 3: Deep Investigation
Use the ToT loop to investigate one hypothesis at a time.
#### Forming Hypotheses and Investigation
1. Each hypothesis must be provable or disprovable.
2. Each hypothesis must directly help explain the original bug.
3. Hypotheses should be independent, clear, and specific.
4. New hypotheses must be more specific than the previous thinking path.
#### Context available to you
You see the full history on the current path, while other branches are visible
only through hypothesis IDs and short summaries.
#### Diagnosing Strategies
{Distributed-systems diagnosis rules and heuristics omitted here for brevity.}
#### Diagnosis Completeness and Self-Critique
Before concluding, you must:
1. trace the full causal chain to code,
2. use critique feedback if available,
3. argue against the current diagnosis,
4. check for multiple contributing blocks.
```

(c) Deep-investigation prompt

Figure 7: Prompt skeletons for the CIRDAN diagnosis phases.

An optional offline utility surveys the checked-out source for up to ten iterations to produce an architecture reference containing a component map, distinctions between easily confused subsystems, and lifecycle entry points. The benchmark loader supplies this reference only when one has been generated for the repository.

C.5 PROMPTS AND SAVED ARTIFACTS

Diagnosis guidance. Condensed skeletons of the survey, triage, and deep-investigation prompts appear in Figure 7. The default system prompt asks the model to use issue artifacts as evidence

while questioning incomplete descriptions and inferred preconditions. It directs the agent to trace state through initialization and cleanup, identify the component responsible for lifecycle decisions, and inspect callers when a function appears to consume or report already faulty state. For omission bugs, the diagnosis should identify where the missing operation belongs. Before concluding, the model is asked to connect the causal steps to code locations and consider which part of the report its explanation leaves unanswered. Hypothesis-generation guidance asks for broad component alternatives followed by increasingly precise investigation plans. The harness itself enforces the phase limits and acceptance rules described above.

Auxiliary calls and saved artifacts. Hypothesis proposal, leaf-node scoring, optional causal critique, and deep-stage completeness checking use separate schema-constrained calls. A run stores its structured diagnosis, model and tool histories, thought-tree snapshot, verification decisions, termination reason, and recorded usage statistics. When an offline architecture reference exists, it is inserted as repository context.

D Design Space and Lessons

This section records design choices explored during development. Several are not part of the final system; they explain why CIRDAN uses a single agent with explicit external state instead of a more elaborate harness.

Single Agent vs. Multi Agent. A fundamental design question is whether distributed debugging should rely on a single large agent or a pipeline of specialized agents. Triangle and OpenRCA adopt multi-agent architectures, on the intuition that separate agents can specialize and reduce context size (Yu et al., 2025a; Xu et al., 2025). Coding agents such as Codex CLI and Claude Code instead use a single agent with a rich internal loop.

Bug localization appears to benefit from specialization because hypothesis generation, evidence collection, and hypothesis refinement involve different skills. In early prototypes, we therefore experimented with a more decomposed architecture in which these stages were more sharply separated. The main benefit was that each stage operated within a narrower context and was less likely to overflow during long investigations.

However, this separation also introduced a clear drawback: information loss at stage boundaries. Since each phase received only the summarized output of the previous one, important details could disappear. For example, a triage-stage summary might correctly implicate a file yet omit the precise evidence needed for a later deep dive to recover line-level citations. This observation pushed us toward the current design, where the runtime stays unified and the harness maintains explicit branch-local state.

The lesson was that diagnosis is sensitive to information lost between stages. CIRDAN therefore keeps one agent and places structure in the harness and the tree state; §4.4 measures the same trade-off with current models.

Reasoning Mechanism. ReAct (Yao et al., 2023b) lets an agent repeatedly invoke tools and reason until the goal is achieved or the budget is exhausted. We also experimented with CIRDAN-*Naive*, a ReAct agent using the same tools and prompts as CIRDAN. We observed that CIRDAN-*Naive* performed well on bugs with clear information in the issue description (e.g. the buggy function is already in the stack trace attached in the description). When bugs become more complicated, or there is no direct implication in the description, CIRDAN-*Naive* often sticks to a single thought and falls into tunnel vision. Recent research (Majgaonkar et al., 2025) suggests similar findings.

CIRDAN addresses this problem by implementing structured hypothesis exploration inspired by tree-of-thought. In the original version, we did not adopt heuristic node selection, but used DFS instead (i.e., after expanding a node with new hypotheses, select and investigate the first child). Since we require the hypothesis to be proved or disproved only with solid evidence, the agent is conservative in performing a backtrack. This resulted in similar tunnel-vision behavior in the naive DFS implementation of CIRDAN: the agent tended to dig deep into one branch and ignore other plausible but shallower alternatives. We later adopted heuristic selection among leaf nodes based on the intuition that branches generated under tunnel vision would often be less relevant to the original bug than

other unexplored hypotheses generated earlier under broader context. Structured hypothesis exploration with heuristic selection ultimately performed better on the more complex bugs that defeated the plain ReAct agent CIRDAN-*Naive*.

Tools Selection. We initially exposed rich IDE-like capabilities, including symbol lookup, definition and reference search, documentation retrieval, and AST navigation. Although these tools are more powerful than shell commands, they consistently underperformed. The agent rarely invoked the richer interfaces even with explicit prompting, and often attempted to emulate simple commands such as `rg`, `ls`, and `cat` through the IDE tool itself. Accuracy dropped by roughly eight percentage points in line-level metrics compared to a plain shell baseline.

This behavior reflects the model's post-training distribution. Large language models are more familiar with shell patterns than with IDE or LSP abstractions. Rich tools therefore introduce latency and cognitive overhead without improving retrieval quality. As a result, CIRDAN exposes retrieval functions primarily through lightweight shell commands, with `view_symbols` added as a targeted complement for symbol enumeration.

Prompt Tuning. We found it important to use a dedicated prompt for the root node of the hypothesis tree. At the root, the "hypothesis" is just the original issue description, and the first round of hypothesis generation determines the entire search space. A carefully tuned root prompt produces broader and more informative hypotheses and improves downstream exploration.

We also compared multi-stage reasoning with single-stage reflection for the investigation loop. The multi-stage design used an explicit checklist of filtering evidence, reconstructing partial executions, and proposing successor hypotheses. The single-stage design supplied only debugging hints and asked the model to perform all reasoning in one step. Multi-stage reasoning helped earlier, weaker models such as GPT-4o, but introduced unnecessary steps and higher latency for stronger reasoning models such as o4-mini. For these models, single-stage reasoning was sufficient and more stable.

These observations suggest that prompt structure should be adapted to the reasoning capabilities of the model, and that additional stages do not necessarily yield better results.

Remaining Limitations. Inspecting the trees produced by earlier configurations revealed two recurring problems:

a) In some bugs, CIRDAN repeatedly generates hypotheses that point toward the correct region. But two challenges remain. Heuristic node selection may still choose the wrong branch to explore and waste substantial time. Moreover, even if the correct node is selected, it can remain difficult for the agent to confirm the hypothesis with solid evidence, especially when the buggy code is not obvious.

b) Even with heuristic node selection, CIRDAN may still take a long time to find the correct location, or even fail and drift in the wrong direction. This can happen if the bug is not well-described, or the observed failure is "far away" from the buggy component.

Both cases show a limitation of our current design: looking merely at the code is insufficient to determine whether there is a bug, or to confirm if the search direction is correct. This is a difficult task to even a human developer, who usually leverages environmental feedback to guide the debugging process, e.g., add some print statements, run the system and then check the log. However, bug reproduction is another nontrivial software engineering task. Our agent can be improved with the help of environmental feedback.

E MAIN EVALUATION PROTOCOL

Task and reference diagnoses. Given a report, a repository at a specified revision, and supplied issue artifacts, the agent must explain the failure and identify the responsible code. The explanation should connect the triggering conditions to the observed symptom. Logs, tests, and program analysis can supply evidence when available; we do not assume a reliable reproducer or access to the affected deployment. We manually reviewed the reference diagnoses and task inputs to check that the reports, source code, and artifacts provide sufficient evidence to identify the cause.

Table 5: Systems in the evaluation. Blocks are annotated root-cause code blocks; multi-block and cross-file count bugs whose cause spans several blocks or files.

System	Language	Bugs	Blocks	Multi-block	Cross-file
Kafka	Java, Scala	8	18	5	4
Elasticsearch	Java	4	8	2	1
Ray	C++, Python	4	7	2	2
HBase	Java	3	4	1	0
HDFS	Java	3	9	2	1
TiKV	Rust	3	8	1	1
ZooKeeper	Java	3	4	1	0
Ignite	Java	2	7	2	2
Ignite 3	Java	2	5	2	1
Kubernetes	Go	1	2	1	0
etcd	Go	1	1	0	0
Total		34	73	19	12

Table 6: Issue-report evidence and difficulty signals for the 34 evaluated bugs.

Statistic	**Value**
Available Evidence	
Contains stack traces	14 / 34 (41%)
Contains logs	11 / 34 (32%)
Names at least one code identifier	28 / 34 (82%)
Mentions an explicit causal cue	5 / 34 (15%)
Difficulty Signals	
No stack-trace anchor	20 / 34 (59%)
No explicit causal cue	29 / 34 (85%)
Mentions 3+ code identifiers	16 / 34 (47%)
Long-form prose ($\geq$1000 chars)	21 / 34 (62%)

The workload includes production failure scenarios curated from public issue reports. Developer discussions, fixing changes, and source code establish the reference diagnoses; the agent receives the report, pre-fix source, and supplied artifacts. Prolonged discussions expose alternative explanations, while more routine cases show where agents already diagnose effectively.

Workload. The 34 bugs come from 11 distributed systems and six implementation languages (Table 5); they are drawn from DistDebug-Bench (Anonymous, 2026). Their causes span 73 annotated code blocks: 19 bugs have more than one responsible block and 12 span more than one file. The reports vary in the evidence they provide (Table 6): 14 include stack traces, 11 include logs, and only 5 state an explicit causal cue.

Shared bugs and configurations. Table 1 restricts all 13 model–harness configurations to the same 34 bugs. §4.1 specifies the baselines and the two CIRDAN configurations: the guided configuration, used with GPT-5.2-Codex, and the light configuration, developed with GPT-6, from which the configurations of the other models are derived. Each comparison is between complete harnesses, whose prompts, tools, budgets, and completion handling differ.

Localization and aggregation. Coverage and precision compare reported locations against annotated files, functions, lines, and source blocks. Block coverage counts an annotated block as recovered when a reported source range overlaps it, without a line tolerance; when a bug admits alternative annotations, any one of them counts. Block precision allows a five-line tolerance.

We average each metric across the included attempts within a bug and then equally across bugs. F1 is $2CP/(C+P)$ using the unrounded aggregate coverage C and precision P, with zero when both

are zero. Each configuration has up to three attempts per bug, and recorded total tokens follow the same aggregation.

F Threats to Validity

The workload covers public bugs from a limited set of distributed systems, which limits generalization to other diagnosis settings. Public availability also permits prior model exposure even when Internet access is disabled during evaluation. Localization scores measure agreement with annotated code: they can miss valid alternative causal boundaries or reward broad reported ranges, and do not by themselves establish a correct causal explanation. The configurations were developed on benchmark cases; their adaptation is model-specific and adds no benchmark-specific rules (§4.1). Historical comparisons differ in execution policies or provider behavior. We interpret results within their recorded cohorts and configurations; Appendices K and L describe the leakage controls and the adaptation of the configuration to GPT-6.

G Supporting Trace Analysis

G.1 Study Coverage and Trace Selection

The task and reference diagnoses are described in Appendix E; block and file coverage follow the definitions in §4.1. The main-text statistics use the 34 shared bugs in Table 1 and its seven baseline configurations other than Gemini–Mini. Their file coverage exceeds block coverage by 16.1–28.0 percentage points; the largest gap is Mini-SWE-agent with GPT-6-Astra (88.8% versus 60.8%). As in the main evaluation, averages weight bugs equally.

The early-scope comparison uses the three Mini-SWE-agent configurations: GPT-5.2-Codex, GPT-6-Astra, and DeepSeek-V4-Flash. On the 34 bugs, these have 102, 102, and 101 scored diagnoses, respectively, with 86, 47, and 101 eligible early windows. Their mean block coverage, averaged within each bug and then across bugs, matches Table 1; the averages include scored attempts that are ineligible for the early-scope analysis. Eligibility is determined without the final localization outcome. Appendix K describes environment preparation and exposure analyses.

G.2 Early Search Scope and Subsequent Exploration

This analysis supports the early-inspection comparison in §2. Appendix A.3 gives a two-attempt example with GPT-5.3-Codex (OpenAI, 2026a).

Early hits. An attempt has an early hit if one of its first two source-inspection calls returns content from a ground-truth file, that is, a file containing an annotated causal block. A source-inspection call is a file read or a search that returns code snippets; directory listings, path-only search results, empty results, and failed reads are not counted. All files returned by one call share its position, and repeated reads count as separate calls. We group completed attempts by final block coverage (zero, partial, or full) and compute the early-miss rate in each group. Attempts with fewer than two qualifying calls, incomplete traces, or execution failures are excluded, as are traces whose early window cannot be attributed to individual files, for example because of compound or wildcard reads.

Table 7 gives per-configuration rates. Pooled over eligible attempts, 45/72 (62.5%) attempts with zero coverage begin with an early miss, versus 27/94 (28.7%) attempts with full coverage. The zero-coverage early-miss rate ranges from 16.7% to 80.0% across configurations. Exclusions differ by model. For GPT-5.2-Codex, 16 scored traces have an ambiguous early window, and 27 failed or unaccepted runs are excluded from both scores and rates. For GPT-6-Astra, 54 traces have an ambiguous early window and two have fewer than two qualifying calls, with one trace in both groups. DeepSeek has one failed or unaccepted run and 101 eligible scored attempts. Because exclusions depend on model behavior, the pooled rates describe the eligible traces only.

Exploration after an early miss. For each early-miss attempt, we follow the remaining tool calls and record three events: a source search whose scope widens to an ancestor of an earlier search scope; a source read in a directory not visited before; and a source read that returns to a directory

Model	Eligible / scored	Early misses / eligible attempts (%) Zero coverage	Partial coverage	Full coverage
GPT-5.2-Codex	86 / 102	19/30 (63.3)	7/25 (28.0)	10/31 (32.3)
GPT-6-Astra	47 / 102	2/12 (16.7)	0/15 (0.0)	0/20 (0.0)
DeepSeek-V4-Flash	101 / 101	24/30 (80.0)	8/28 (28.6)	17/43 (39.5)
Pooled	234 / 305	45/72 (62.5)	15/68 (22.1)	27/94 (28.7)

Table 7: Early file inspection in the Mini-SWE-agent configurations from the main evaluation, restricted to its shared 34 bugs. An early miss means neither of the first two source-inspection calls returned code from a ground-truth file. Columns separate final source-region coverage; only traces with unambiguous early windows enter the rate denominators.

after a visit elsewhere. Paths are normalized to repository-relative locations using the command's working directory, and a bare `cd ..` or `ls` is not counted. We also record when the attempt first reads a ground-truth file, which is distinct from localizing the cause in the final diagnosis.

Among early misses, the agent later reads a ground-truth file in 32/36 GPT-5.2-Codex attempts, 1/2 GPT-6-Astra attempts, and 49/49 DeepSeek attempts. At least one widened search or directory revisit occurs in 34/36, 1/2, and 49/49 attempts, respectively. Every later call can be attributed to a search scope in only 27, zero, and 24 of these attempts, so the event counts are lower bounds. Most early misses thus reach a ground-truth file later in the investigation.

G.3 Report Anchors and Investigation Depth

We review whether a completed investigation follows an anchor in the report and finalizes a plausible explanation while a relevant alternative remains unexamined. The review records the report anchor, inspected code, explanation, unresolved evidence, and stopping decision. Following an anchor can lead directly to the cause, and a localization miss alone does not indicate premature commitment: as Appendix A.2 shows, an agent can describe the memory–disk mismatch while reporting different responsible code. The ZooKeeper-2845 trajectories illustrate this separation between recognizing the broad failure mechanism and attributing it to the relevant lifecycle code.

Further observations of hypothesis comparison, verification, and attribution changes appear in §4.3 and Appendix H, which examine what the agent reports after it reaches relevant code.

H Structured versus Linear Investigation

Protocol. This comparison uses five fixed bugs, with two fresh attempts per bug in each arm: `HBase-28376`, `Kafka-7044`, `Kafka-8289`, `etcd-17507`, and `HDFS-10512`. Structured CIRDAN uses medium reasoning effort; linear CIRDAN uses xhigh. Both arms share GPT-6, reports, pinned source, injected context, tools, diagnosis guidance, critique availability, and per-attempt limits, so the arms differ in both workflow and reasoning effort. All 20 original attempts are included, with no replacements.

Table 8: Focused workflow comparison. Coverage and precision are percentages. The primary row retains two unaccepted linear attempts as zeros; the second row removes both sides of those pairs.

Cohort	Pairs	Block coverage Structured	Linear	Block precision Structured	Linear
All attempts	10	70.8	58.3	76.7	51.7
Both finals accepted	8	76.0	72.9	70.8	64.6

Results and incomplete attempts. Eighteen attempts return accepted final diagnoses (Table 8). Coverage uses source-line overlap with a fixed file-equivalence rule; precision uses a five-line tolerance. Both linear attempts on `Kafka-8289` time out during critique. Their drafts already identify the

same root block and missing-inactivity-gap mechanism as the structured answers, but remain unaccepted and receive zero in the primary analysis, so part of the all-attempt difference comes from completion failures. On the eight valid pairs, structured investigation uses 2.57× the known cost; usage of the unanswered calls is unknown.

Observed investigation behavior. All ten structured surveys already contain the principal final mechanism. The runs generate and triage 21 hypotheses, then each deeply follows one branch. Across 42 primary deep responses, there are no deep branch switches, revisits, or second-level expansions. On `HBase-28376`, an early proposal names startup and a descriptor helper before the final selects migration. The survey already identifies the missing destination column family, and the later correction concerns attribution. Linear investigation also identifies migration, and the second pair reports the same block. On `HDFS-10512`, deeper work checks the independent invalidation thread and exception path after the survey has already described the race.

Sensitivity to reported ranges. All four `Kafka-7044` answers explain how mixed metadata availability drops partitions, bypasses retries, returns a partial offset map, and leads to null unboxing. The first structured answer additionally reports the half-open range $[416, 423)$ in `fetchOffsetsByTimes`, touching the one-line annotation $[422, 423)$. Shortening that range to $[416, 421)$ preserves the described early-return code and removes the entire valid-pair coverage advantage; the reported scores use the recorded outputs. The second pair reports identical ranges. On these five cases, the structured arm's gains come from completion and reported ranges, and deep branching is rarely exercised.

I Reasoning-Effort Comparison

This comparison tests whether raising Codex's reasoning effort improves localization on five bugs, and compares the result with CIRDAN.

Results. Across two attempts per bug, increasing Codex's effort from medium to xhigh raises block coverage from 46.7% to 50.0%, while precision changes from 43.3% to 41.7%. Recorded total tokens increase by 1.44× and estimated cost by 1.50×. CIRDAN at medium effort obtains 65.8% coverage and 71.7% precision. Nine of the ten CIRDAN attempts finish during the survey. The principal failure mechanisms appear in the explanations of all three conditions; label alignment and alternative repair boundaries account for much of the localization difference.

Scope and selection. The comparison contains five bugs from the benchmark: `HBase-28376`, `Kafka-7044`, `Kafka-8289`, `etcd-17507`, and `HDFS-10512`. Each has two attempts in three GPT-6 conditions: Codex CLI at medium effort, Codex CLI at xhigh, and CIRDAN at medium. The medium-effort Codex and CIRDAN attempts come from earlier runs, giving 20 existing attempts and ten new xhigh attempts; third medium-effort attempts form a separate sensitivity analysis. We selected bugs where the baseline had headroom, without requiring a previous CIRDAN win, and fixed the selection before inspecting the new outputs. `HBase-28376` admits an alternative repair boundary, and the `Kafka-7044` annotations omit changed root-cause helpers, so low overlap on these bugs can accompany a correct causal explanation.

Matched inputs and fixed attempts. The conditions use the same reports, source revisions, and issue artifacts. The new Codex runs keep the earlier task prompts, base instructions, CLI version, and GPT-6 alias and provider route, and change only the requested effort to xhigh. An offline wire check confirms that the CLI sends xhigh; the effort the provider applies is not observable. The CIRDAN attempts keep their original execution policies. All final diagnoses use one scorer and label revision, and evaluator artifacts are excluded from diagnosis inputs.

The time limit is 1,800 seconds per attempt, enforced across checkout and CLI activity. New attempts run serially, and dispatch pauses after a completed attempt reaches $10 estimated cost or cumulative known cost reaches $50; these are review thresholds, with no hard dollar cap. Operational failures or missing accounting also pause dispatch. All dispatched failures are reported, with no replacement retries or best-attempt selection.

Table 9: Focused ablation on the 12-bug overlap across all setups. R removes static routing from the full system. H1–H3 are harness ablations, P1–P3 are prompting ablations, and S1–S2 are supplementary controls. We report localization quality together with run-to-run variability for block coverage over three independent attempts per bug.

Setup	Group	Block Cov.	Block Prec.	Mean Spread	Flaky Rate
Full	Reference	49.3	29.6	0.375	0.417
R: no static routing	Reference	46.9	29.4	**0.204**	**0.250**
H1: no arch doc	Harness	46.4	25.3	0.329	0.417
H2: no shell ctx	Harness	52.1	30.6	0.375	0.417
H3: no ctx injection	Harness	**52.6**	**34.3**	0.517	0.500
P1: no prompt rules	Prompting	50.6	28.9	0.454	0.500
P2: no critique	Prompting	41.0	27.8	0.292	0.333
P3: no anchor	Prompting	36.7	24.4	0.496	0.583
S1: pure ToT	Supplementary	41.2	33.3	0.500	0.583
S2: Phase-1 only	Supplementary	46.5	30.6	0.482	0.667

J Additional Evaluations

This section reports evaluations that use earlier annotations and harness configurations: the earlier 37-bug comparison with GPT-5.2-Codex, phase statistics from CIRDAN traces, and mechanism comparisons on a 12-bug subset.

J.1 Per-Case Localization and Reliability

On the earlier 37-bug workload with GPT-5.2-Codex, four bugs have zero block coverage under all three baselines and nonzero coverage with CIRDAN: `ZooKeeper-2845` (1.000), `Ray-22122` (0.667), `HBase-28962` (0.333), and `Ray-14264` (0.111). On `TiKV-18498`, CIRDAN obtains 0.667 coverage versus 0.333 for Mini-SWE-agent and OpenCode. These scores use the original annotations. Scores also vary across attempts: 40.5% of bugs have non-identical CIRDAN block scores across attempts, versus 27.0–29.7% for the baselines.

J.2 Investigation Traces and Phase Use

A phase study of 111 attempts with an earlier stateful-critique configuration records 108 returned diagnoses: 56 during triage, 52 during deep investigation, and none during the survey. The saved trees average 6.03 nodes including the root, a maximum depth of 1.31, and 3.53 initial hypotheses.

In a representative `Ray-22122` attempt, CIRDAN proposes three hypotheses: worker backlog accounting, raylet resource reporting, and autoscaler aggregation. All three are triaged. The raylet hypothesis attributes the failure to a limit on the number of scheduling classes included in a resource report; the agent rules it out because the reported workload uses a single CPU shape. The worker-accounting branch then receives deep investigation and yields the final diagnosis. Comparing the hypotheses after reading code eliminates a plausible branch.

J.3 Mechanism Comparisons

On a 12-bug subset, full CIRDAN obtains 49.3% block coverage, compared with 46.5% for a Phase-1-only control and 41.2% for a pure tree-of-thought control. Against Phase 1 only, full CIRDAN wins on three bugs, ties on five, and loses on four; most of the difference comes from `Ray-22122`, where full CIRDAN obtains 66.7% coverage and Phase 1 only obtains zero. Mean recorded tokens per attempt are 2.22M, 1.35M, and 0.76M, so the controls also spend less inference.

Removing single mechanisms gives mixed results. Relative to an earlier 46.9% reference, removing critique lowers coverage to 41.0% and removing anchor guidance lowers it to 36.7%, whereas removing shell context, all context injection, or prompt rules raises the mean on this subset. Removing all context injection obtains 52.6% coverage, compared with 49.3% for full CIRDAN, but raises the mean per-bug spread from 0.38 to 0.52, where spread is the difference between the highest and

Information supplied	File coverage	Function coverage
Report and repository	75.2%	45.5%
Report with issue identifier, closed-book	43.9%	17.3%
Report with identifier redacted, closed-book	43.2%	21.4%
Project and issue identifier only, closed-book	0.0%	0.0%

Table 10: Earlier GPT-5.4 contamination probes on 66 bugs whose reports do not name a gold file or function, with three attempts per bug. Closed-book conditions provide no repository, tools, Git history, or Internet access. Repository-assisted results use the same bug subset.

lowest coverage among a bug's three attempts. The variants were not budget matched. The table keeps the original configuration names, some of which refer to mechanisms of an earlier runtime, including older static-routing and prompting variants.

K Environment Preparation and Leakage Controls

The 34 bugs are drawn from DistDebug-Bench (Anonymous, 2026), and the analyses in this section come from its evaluation on the full 120-bug workload. Each task supplies the incident evidence and the pre-fix repository; during benchmark construction, every case was verified to be diagnosable from these inputs. Agents could otherwise find the target issue through the repository history, for example with `git log | grep 2845`, so the Git history is removed from the workspace. At run time, a leakage guard scans tool calls, tool outputs, and visible agent reasoning for the target issue identifier. Flagged trajectories are discarded and rerun until each bug has three unflagged attempts. For GPT-5.4, the guard flagged 102/360 (28.3%) trajectories before Git-history removal and 4/360 (1.1%) afterward.

K.1 Public Bugs and Possible Prior Exposure

The workload includes longstanding public bugs: ZooKeeper-2845, for example, was reported in 2017 (Apache ZooKeeper, 2017). A model may therefore know a bug from pretraining even when it cannot retrieve it at run time. The DistDebug-Bench evaluation examines this possibility by correlating scores with public exposure and by diagnosing reports without repository access.

Issue age and public discussion. Across the 120-bug workload, GPT-5.4 block coverage has near-zero correlation with issue age (Pearson $r = -0.002$, Spearman $\rho = -0.031$) and issue-comment count ($r = -0.052$, $\rho = -0.048$): older or more extensively discussed issues are not easier in this evaluation.

Closed-book diagnosis and identifier recall. The closed-book study evaluates GPT-5.4 on 66 bugs, excluding 54 reports that name a gold file or function. Each closed-book condition supplies no repository, tools, Git history, or Internet access, and uses three attempts per bug. Table 10 compares these conditions with repository-assisted diagnosis on the same subset. Function coverage is 45.5% with repository access, 17.3% with an identifiable report alone, and 21.4% when the report's issue identifier is redacted. Prompts containing only the project and target identifier obtain zero file and function coverage. A separate report-to-identifier probe recovers none of the 360 exact target identifiers.

Repository-assisted localization is substantially stronger than closed-book localization, and disclosing the issue identifier does not improve function coverage, so direct identifier recall does not explain the repository-assisted scores. Nonzero closed-book coverage can reflect general knowledge of project structure and clues remaining in the report. These probes cover the tested models; we did not repeat them for GPT-6.

Failures on these bugs remain informative: even if a model has seen a bug before, it can commit to an incomplete explanation and fail to revise it.

L ADAPTING THE CONFIGURATION TO GPT-6

The light configuration adapts the guided one to how GPT-6 investigates. We ran GPT-6 with the guided configuration, developed with GPT-5.2-Codex, on nine of the evaluated bugs with three attempts each, and compared it with GPT-5.2-Codex under the same configuration and with GPT-6 under the light configuration on the same bugs. This earlier run was scored with the evaluator's tolerant block coverage, so all three use that measure here.

Under the guided configuration, GPT-6 ends its survey early: it makes 10.9 tool calls there on average, against 14.2 for GPT-5.2-Codex, with a budget of 15. It then spends the run in deep investigation and exceeds the 1,800-second budget in 6 of 27 attempts. The 21 completed attempts reach 71.4% block coverage and 73.8% precision; counting the timeouts as failures, coverage is 55.5% and precision 57.4%. Under the light configuration, with a smaller survey budget and stricter acceptance, GPT-6 completes all 27 attempts with 64.8% block coverage and 64.8% precision, using 7.4 survey calls and 6.2 later calls on average, against 10.9 and 16.4 under the guided configuration. GPT-5.2-Codex under the guided configuration reaches 38.9% coverage and 25.3% precision on these bugs.

GPT-6 reaches a plausible diagnosis within far fewer steps than GPT-5.2-Codex, so the guided configuration's survey budget and guidance do not match how it investigates: the survey ends before the budget is used, and the remaining effort goes into long deep investigations that often exceed the time budget. The light configuration shortens the survey and moves the check on a diagnosis to its acceptance.