

Evaluating Agentic Code Repair Capabilities in Distributed Systems

Yibo Yan Huijuan Wang Junzhou He Yizhuo Liang
 Shaoyu Wang Huanchen Sun Seo Jin Park
 University of Southern California
 {yiboyan, huijuanw, junzhouh, yizhuoli,
 wangshao, huanchen, seojinpa}@usc.edu

Abstract

LLM-based coding agents have advanced rapidly on single-process SWE tasks, with frontier models now clustering in the high-70s on SWE-bench Verified. Distributed-system debugging, however, remains an under-explored regime: bugs span processes, nodes, and protocol interactions, with root causes rarely recoverable from source alone and brute-force exploration intractable across non-deterministic interleavings. This leaves two gaps in LLM and agent evaluation: no code-repair benchmark targets distributed-system bugs, and no controlled study isolates how much externally provided debugging context changes agent success on them. We introduce DDBENCH, a code-repair benchmark of 60 historical bugs mined from 13 open-source distributed systems, partitioned into three difficulty tiers. DDBENCH evaluates every case under two matched conditions: a *symptom-only* condition where the agent receives only the bug symptom and repository, and a *context-augmented* condition where it additionally receives bounded debugging context (logs, traces, runtime state, and targeted code-investigation notes), isolating the effect of debugging context from model capability. The evaluation of 10 LLMs on DDBENCH reveals several findings. First, distributed debugging exercises a reasoning dimension that single-process benchmarks do not surface: models’ pass rates span 61 pp, and pairwise bootstrap separates 9 of 15 top-tier model pairs at $p < 0.05$ on DDBENCH’s hardest case-set. Second, bounded debugging context lifts aggregate pass rate by +18.1 pp, and the lift is asymmetric: weaker models gain pass rate, while stronger models gain efficiency. Third, debugging context requires careful curation, as even faithful debugging context can sometimes mislead LLMs.

1 Introduction

Large Language Models (LLMs) and LLM-based coding agents have advanced rapidly on software engineering tasks [14, 31, 24, 26, 21, 9]. Frontier models now routinely resolve multi-file bugs even through simple agent harnesses [27, 30], and SWE-bench Verified [13] provides an established test set and evaluation protocol for tracking continued progress in this regime.

Debugging distributed systems, however, remains an under-explored regime for LLM-based agents, and is substantially harder than single-process code repair. A distributed-system bug rarely surfaces in a single stack trace as its root cause spans processes, nodes, and network paths. Diagnosing such a bug requires reasoning about cross-process causality, non-deterministic interleavings, data races, and protocol-level invariant violations. Consequently, the critical debugging context (information that helps the debugging process) is often hard to surface from static reasoning on the source code alone making logs, traces, and runtime state snapshot valuable assets. Nor can an agent reconstruct critical debugging context by brute-force enumeration and exploration, since a distributed bug often depends on the precise sequence of events across processes (*e.g.*, whether a timeout fires before or after a

reply arrives), and the number of such sequences is intractable. Therefore, existing LLM and agent benchmarks that target single-process repair tasks can poorly capture distributed debugging capability.

This leaves two concrete gaps in LLM and agent evaluation. First, no code-repair benchmark targets distributed-system bugs. Existing code-repair benchmarks (*e.g.*, SWE-bench [13], DebugBench [22], and Multi-SWE-Bench [37]) focus primarily on single-process workloads. IT-operations benchmarks such as AIOpsLab [8] and CloudOpsBench [25] reach across distributed services but evaluate operational diagnosis from telemetry rather than source-level code repair. There is therefore no principled way to compare models on debugging distributed-system bugs at the source-level. Second, no existing evaluation quantifies how much controlled debugging context (*e.g.*, logs, traces, diagnostic code exploration notes) changes an agent’s success rate and behavior in solving these distributed-system bugs, relative to working from the bug symptom and codebase alone.

We introduce DDBENCH, a code-repair benchmark that contains 60 historical bugs mined from 13 open-source distributed systems, partitioned into three disjoint difficulty tiers: tier-1 (31 cases, the most challenging set), tier-2 (15 cases), and tier-3 (14 cases). DDBENCH therefore enables principled measurement of model capability on distributed-system debugging which addresses the first gap.

Since brute-force exploration is prohibitively expensive in this regime, the value of externally-provided debugging context becomes measurable, making the context channel itself a natural experimental knob. Therefore, DDBENCH is designed to run every case under two conditions: (i) a *symptom-only* condition, where the agent receives only the bug symptom and codebase and must surface any additional information through its own exploration; and (ii) a *context-augmented* condition, where the agent additionally receives a curated debug-context bundle along with the bug symptom and codebase. This design allows us to isolate the effect of additional debugging context from model capability, addressing the second gap by providing a controlled experimental setup.

We evaluated 10 proprietary and open-weight LLMs with mini-swe-agent [30] through DDBENCH’s two-condition harness. Two major findings emerge:

(1) Cross-model separation on distributed debugging. Under the *symptom-only* condition on tier-1 case set, 10 models spanned 61 pp in pass rate. Pairwise bootstrap tests ($n = 31$ cases) distinguished 9 of 15 top-tier model pairs at $p < 0.05$. For the six models we sampled with publicly reported SWE-bench Verified scores¹, the same models span only 7 pp on SWE-bench but 58 pp on DDBENCH tier-1 (Figure 1). The contrast is consistent with the fact that distributed debugging exercises a reasoning dimension that is not directly measurable by single-process code-repair tasks.

(2) Additional debugging context lifts pass-rate unevenly and cuts token cost consistently. Adding curated debugging context lifted aggregate pass rate from 32.6% to 50.6% (+18.1 pp). The gains break down differently across models. Weaker models gained primarily in pass rate (GLM 4.7: 9.7% $\rightarrow$ 48.4%). In contrast, stronger models saw only marginal pass-rate gains but substantial cost reductions (Opus 4.6: 69% lower token consumption and 41% fewer steps-to-completion). The gain of adding debugging context was also large enough to reorder model ranking. Specifically, context-augmented GPT 5.4 matched Opus 4.6’s symptom-only pass-rate while consuming 60% fewer tokens. Furthermore, we observed that per-case outcomes are not uniform: a faithful debugging context (*e.g.*, error outputs from a test case that reveals the bug) sometimes can even mislead strong models when its observation sits far from the fault (§3.3), revealing the hidden risk of adding more information without careful curation.

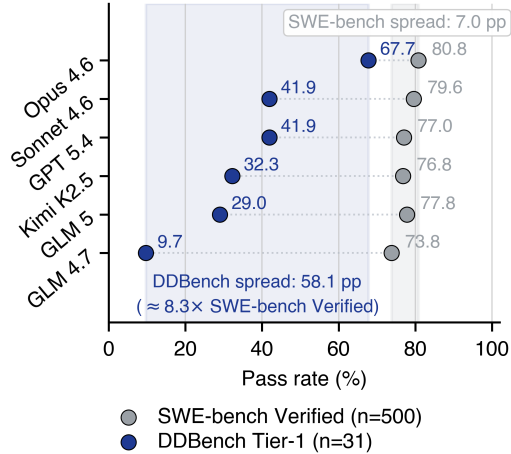


FIGURE 1: DDBENCH sharply differentiates models on distributed debugging. Under the *symptom-only* condition, six models span 58 pp on DDBENCH tier-1 ($n=31$), compared with only 7 pp on SWE-bench Verified ($n=500$).

¹We sampled Claude Opus 4.6, Claude Sonnet 4.6, GPT 5.4, GLM 4.7, GLM 5, and Kimi K2.5. We obtained the score for GPT-5.4 (high thinking) from Epoch AI as OpenAI did not officially release the number; other scores come from model vendors’ official reports.

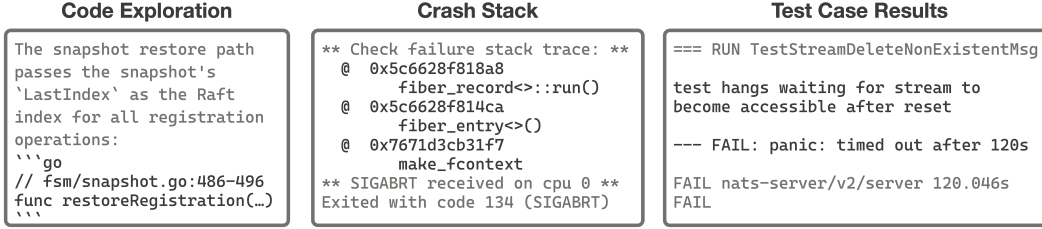


FIGURE 2: Examples of debug-context files shipped in DDBENCH.

The two findings reveal that the same regime that makes curated debugging context valuable also spreads models apart, because the agents that suffer most from missing critical debugging information that cannot be easily compensated through expensive exploration and reasoning. DDBENCH therefore measures not only whether an agent can fix a distributed bug, but also how much adding debugging context helps. One prior work shows that the context curated via lightweight pre-exploration of the target codebase can substantially improve the resolution rate [20]; following this same vein, DDBENCH concretely anchors this insight in the distributed debugging regime. Thereby, we summarize our contributions as follows:

- **DDBENCH:** a benchmark for agentic distributed-system debugging, comprising 60 cases from 13 open-source systems, each independently reproduced, sanitized against leakage, and reviewed by five distributed-systems researchers. The size reflects this quality bar, not a scale ceiling.
- **Model differentiation on distributed debugging:** an evaluation of 10 models that yields a spread of 61 pp on DDBENCH’s hardest case set and separates 9 of 15 model pairs, indicating that distributed debugging exercises a reasoning dimension different from single-process code repair.
- **The effect of debugging context on agent performance:** a symptom-only vs. context-augmented study that isolates debugging context from model capability. Bounded context disproportionately lifts models’ pass rate while uniformly reducing cost across all models.
- **An extensible testbed for agent-tool research:** DDBENCH is agent-agnostic, and its debugging context channel is a drop-in slot for alternative debugging tools (*e.g.*, tracers, static analyzers, code-search tools), enabling head-to-head comparison of future agentic tool proposals on the same cases and the same capability baseline.

2 DDBENCH

DDBENCH is a code-repair benchmark containing 60 historical bugs from 13 open-source distributed systems, partitioned into three disjoint tiers (tier-1/tier-2/tier-3) ordered from highest to lowest difficulty. For each case, we localize the faulty commit from the upstream issues and PRs. Each case ships with a *git worktree* pinned to the faulty commit, a *symptom description* of the fault (SYMPTOM.md) derived from the original GitHub issue/PR, a hidden *oracle* returning PASS/FAIL on the patched codebase, and a *debug-context* bundle (§2.1). DDBENCH is agent-agnostic, so that any agent that can read a repository and propose a patch can be evaluated.

2.1 Debug Context: What It Is and How It Is Supplied

Symptom description merely describes the observation of the fault without providing additional debugging information, such as logs, traces, or targeted code exploration on the faulty code. We treat such additional debugging information an agent receives, referred as *debug-context*, as a first-class experimental variable. A debug-context is any bounded, per-case artifact that a debugging tool could realistically produce for developers. This covers both *runtime signals* (*e.g.*, process and network logs, stack traces, race-detector reports, partial runtime-state snapshots) and *static-but-targeted code exploration* on the fault (*e.g.*, the suspicious call sites and cross-file relationships a static analyzer would surface). Anything a tracer, debugger, invariant checker, or static code analyzer could generate is admissible; however, the fix itself and any direct pointer to it are excluded. In this DDBENCH release, each case’s debug context bundle contains one to three short Markdown files, median 321 estimated tokens (max 1,403) for models to consume. Figure 2 describes three examples of debug-context files included in this release of DDBENCH.

Evaluation procedure. The evaluation procedure (Figure 3) is a two-condition test, where the agent diagnoses and repairs the same case with and without the debug-context bundle. In the symptom-only

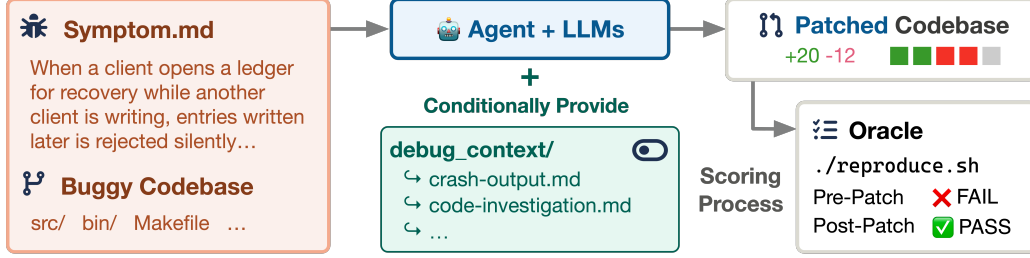


FIGURE 3: The flow of evaluating a DDBENCH bug case. The buggy codebase and agent are containerized together, and the agent can access the codebase and a Markdown symptom description. In the context-augmented condition, the agent also has access to a debug-context bundle (green). After the agent concludes its patch, a hidden oracle runs against the codebase and returns PASS/FAIL as the final evaluation.

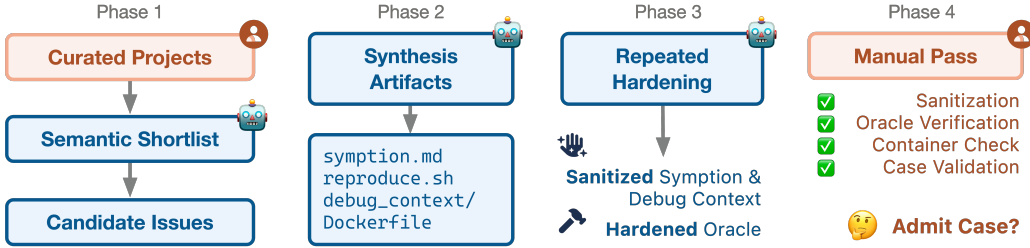


FIGURE 4: The DDBENCH curation pipeline, where an LLM agent performs first-pass mining and artifact synthesis under tight specifications, and human researchers perform final sanitization, validation, and acceptance.

condition, the agent’s container is mounted with the faulty codebase and SYMPTOM.md. The agent is free to read the repository, run commands, and invoke tools its scaffold provides to gather additional context on its own. In the context-augmented condition, the same agent on the same case additionally receives the debug-context bundle, mounted in the containerized codebase with additional prompts to use it. Agents have the same tool call access in both conditions.

A parameterized extensibility slot. The bundle is designed to be a drop-in slot. Namely, a researcher who builds a new structured tracer, debugger, invariant checker, or code-search aid can re-populate debug-context files for cases from their tool’s output and evaluate that tool against any DDBENCH-compatible agent without benchmark re-engineering. What we ship is therefore just one default bundle; alternative bundles produced by other tools can be dropped into the same slot and compared head-to-head on identical cases and oracles. The slot establishes *which debug context an agent sees* as a first-class design axis for agent tool augmentation.

2.2 Dataset Construction

We construct DDBENCH through the semi-agentic pipeline (Figure 4): an LLM agent performs first-pass mining (phase 1), artifact synthesis (phase 2), and iterative hardening (phase 3) under tightly scoped specifications, and human researchers perform final validation, sanitization, and acceptance (phase 4). A candidate is admitted only if (a) the bug was confirmed and fixed upstream and (b) its diagnosis demonstrably requires cross-process or interleaving-aware reasoning. Purely manual curation does not scale to the per-system protocol expertise each case demands, and naive issue scraping cannot discriminate cross-process bugs from single-process ones; the agent is what makes this admission bar enforceable at scale. The mining protocol, symptom and debug-context synthesis, and answer-leakage sanitization are deferred to Appendix §C.

Oracle reliability and manual review. Each oracle is iteratively hardened by an agent with a specialized prompt until it reliably reports FAIL on the faulty commit and PASS on a correctly patched codebase. Cases that fail to stabilize after five hardening iterations are dropped. Oracles accept any patch under which the reproducer reports PASS, not just patches that match the upstream fix textually. A fix-specificity scan additionally flags fix-overfit oracles (*e.g.*, ones that pattern-match a specific code substring or use reflection-based assertions) for manual hardening (Appendix §C.4). Five

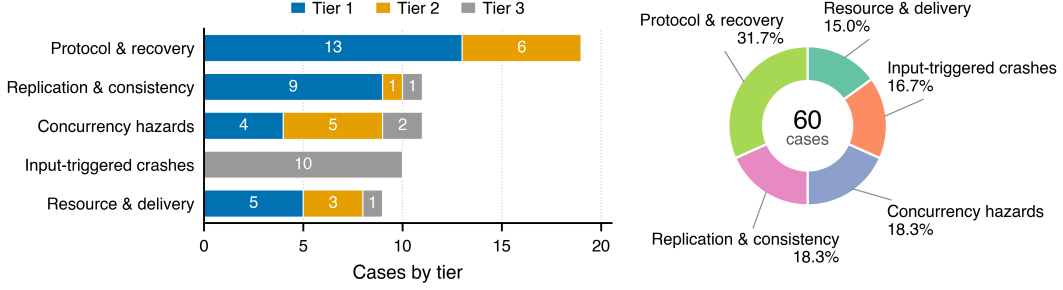


FIGURE 5: Bug-family composition of DDBENCH. **Left:** per-tier breakdown where tier-1 concentrates protocol/recovery and replication/consistency bugs; tier-3 absorbs all input-triggered crashes, which can be localized within a single process; tier-2 bridges the two. **Right:** each family’s overall share of the benchmark.

distributed-systems researchers then manually review every surviving case for quality and re-check symptom and debug-context sanitization on top of the automated passes before final admission.

Dataset tiering. We partition all cases into three disjoint tiers (tier-1, tier-2, tier-3), ranking from the highest to lowest reasoning load and difficulty. Tier-1 is the most challenging set, as it concentrates the hardest cross-process, undeterministic timings reasoning scenarios. The demanded reasoning load decrease from tier-1 to tier-3. Tier-3 cases are bugs in distributed systems that can be localized into single-process misbehaviors without much complex cross-process reasoning. Both tier-2 and tier-3 enable us to understand how tier-1 patterns generalize as the root-causing difficulty decreases.

2.3 Features

DDBENCH covers these system categories: key-value stores (22 cases), consensus libraries (21), messaging systems (10), service meshes (3), streaming engines (3), and a storage engine (1). These cases cover five languages: Go (25), C++ (23), Java (8), Erlang (2), and Rust (2).

Bug families. We group the 60 cases into five families that reflect the reasoning each bug demands rather than its surface symptom (Figure 5). *Protocol and recovery failures* (31.7%) are the largest family: bugs in selecting a coordinator (leader election), copying state across the cluster (log replication), re-syncing a recovering node (snapshot transfer), and changing node membership (cluster reconfiguration), where correctness depends on multi-step protocol invariants holding across nodes. *Concurrency hazards* (18.3%) cover data races, deadlocks, and livelocks that manifest only under specific runtime interleavings. *Replication and consistency violations* (18.3%) produce divergent state across replicas, *e.g.*, operations that appear out of order to clients (linearizability violations), and lost updates. *Input-triggered crashes* (16.7%) and *resource and delivery anomalies* (15.0%) account for the remaining cases. The first three families require reasoning that cannot be discharged from a single process trace. In other words, the LLM and agent must understand state across nodes or across non-deterministic schedules to reason and localize the fault. The tier breakdown in Figure 5 (left) shows that tier-1 concentrates the cross-process families (13 of 19 protocol/recovery cases, 9 of 11 replication cases), while every input-triggered crash falls in tier-3. The tier-1 headline set is therefore not merely a harder version of the same task, but it is structurally different in the reasoning it requires.

See Appendix §B and §D for more dataset details, including the debug-context-bundle-size distribution and a quantitative analysis of DDBENCH’s reasoning difficulty relative to patch size.

3 Evaluation

We evaluate DDBENCH along three axes: (1) whether distributed-system bugs differentiate frontier models that existing benchmarks compress together, (2) whether bounded debug-context improves agent diagnosis success rate and at what cost, and (3) how these effects vary by bug family, difficulty tier, and individual case.

3.1 Setup

Models. We evaluated 10 models, spanning proprietary and open-weight models: Claude Opus 4.6, Claude Sonnet 4.6, GPT 5.4, GPT 5.4 mini, GLM 5.1 (FP8, self-hosted), GLM 5, GLM 4.7, Gemma 4 (31B, self-hosted), Kimi K2.5, and GPT OSS (120B BF16, self-hosted). For each case, we

TABLE 1: Per-model pass rates (%) on tier-1 (31 cases) under symptom-only and context-augmented conditions. Δ denotes the percentage-point improvement. Nine of ten models improved; one tied; none regressed.

Model	Thinking	Form	Total Cost (\$)			Pass Rate %		
			Sym.	+Context	Δ (pp)	Sym.	+Context	Δ
Claude Opus 4.6	Adaptive High	API	59.4	23.6	-60.3%	67.7	67.7	—
Claude Sonnet 4.6	Adaptive High	API	68.4	34.0	-50.3%	41.9	61.3	+19.4
GPT 5.4	High	API	25.0	21.4	-14.3%	41.9	67.7	+25.8
GPT 5.4 mini	High	API	7.9	6.1	-23.5%	45.2	61.3	+16.1
GLM 5.1 (FP8)	Enabled	Open, Self-Host	23.1	14.6	-36.7%	32.3	50.0	+17.7
GLM 5	Enabled	Open, API	42.3	30.3	-28.5%	29.0	45.2	+16.2
GLM 4.7	Enabled	Open, API	38.5	18.7	-51.5%	9.7	48.4	+38.7
Gemma 4 (31B)	Enabled	Open, Self-Host	4.9	4.5	-8.6%	19.4	38.7	+19.3
Kimi K2.5	N/A	Open, API	11.2	7.5	-32.6%	32.3	38.7	+6.4
GPT OSS (120B BF16)	Enabled	Open, Self-Host	3.7	3.3	-11.5%	6.5	29.0	+22.5

use an identical agent scaffold with tool-use access to the repository. The agent being evaluated runs inside the isolated container and is blind to the test oracle. We bound the agent’s max steps to 450 per case, max token cost to \$5 per case, and max time to 30 minutes per case.

Agent. We used `mini-swe-agent` [1] (the successor of `SWE-agent` [30]) as the agent scaffold, as it provides a structured approach to collect token usage, action trajectories, and intermediate reasoning steps. `mini-swe-agent` has proven effective in solving complex SWE problems [30, 10, 9, 21] and has been widely adopted [19, 34, 28, 17], making it a suitable choice for our evaluation.

Metrics and statistical methodology. We report pass rate (fraction of cases where the agent’s patch passes the hidden reproduce oracle), agent steps, first-edit step, total tokens consumed, and action-type distributions. Every case in tier-1 is evaluated by every model under both conditions, yielding 620 binary outcomes (10 models $\times$ 31 cases $\times$ 2 conditions) with two structural axes: per-case difficulty and per-model capability. Our statistical tests are chosen to respect this structure.

Pairwise model differentiation. We performed a pairwise bootstrap test for six models that have both publicly available SWE-bench Verified scores and also evaluated in our study (Opus 4.6, Sonnet 4.6, GPT 5.4, GLM 5, Kimi K2.5, and GLM 4.7). For each of the $\binom{6}{2} = 15$ model pairs, we perform a paired bootstrap over the 31 cases (10,000 resamples) to estimate the pass-rate difference with 95% CI. The SWE-bench Verified panel is computed analogously, but with an unpaired bootstrap over 500 simulated Bernoulli trials per model at the vendor-reported rates, as per-case outcomes are not publicly released for all six models we sampled.

Aggregate debug-context effect across models. We fit a mixed-effects logistic regression $\Pr(\text{pass}) = \sigma(\beta_0 + \beta_{\text{ctx}} \cdot \mathbf{1}[\text{debug-context}] + u_{\text{model}} + u_{\text{case}})$ with random intercepts for model and case, so that the debug-context effect is estimated after controlling for baseline capability (u_{model}) and case difficulty (u_{case}). For the debug-context coefficient we report its estimate $\hat{\beta}_{\text{ctx}}$ with standard error (SE, the estimator’s sampling-distribution standard deviation), 95% CI, odds ratio (OR = $e^{\hat{\beta}_{\text{ctx}}}$, the multiplicative effect of debug-context on the odds of pass), and Wald z -statistic ($z = \hat{\beta}_{\text{ctx}}/\text{SE}$, asymptotically standard normal under the null). We prefer this toolkit over resampling-only alternatives (e.g., a bootstrap over the 10-model population) because 10 units are too few to estimate between-model heterogeneity reliably; the mixed model estimates it parametrically from all 620 outcomes.

3.2 Models Cluster on Single-Process Code-Repair but Separate on DDBENCH

We anchor the differentiation analysis on tier-1 because it concentrates the cross-process reasoning load DDBENCH is built to measure: bugs whose diagnosis hinges on causality across processes, concurrency, and protocol invariants. On the other hand, tier-2 and tier-3 relax this load and serve as robustness checks (§3.5) rather than primary differentiation evidence.

Table 1 reports the per-model pass rates on tier-1 under both conditions. Under the symptom-only condition, the cohort spans 61 pp, with Claude Opus 4.6 at the top (67.7%) and GPT OSS (120B) at the bottom (6.5%). To check that this spread reflects a reasoning dimension SWE-bench does not exercise (rather than a different point on the same dimension), we contrast the DDBENCH results with SWE-bench Verified [13]. Of the 10 models we evaluate, six have published SWE-bench Verified scores: Claude Opus 4.6 (80.8%), Claude Sonnet 4.6 (79.6%), GLM 5 (77.8%), GPT 5.4 (77.0%), Kimi K2.5 (76.8%), and GLM 4.7 (73.8%). These six models cluster within a 7-pp band on SWE-

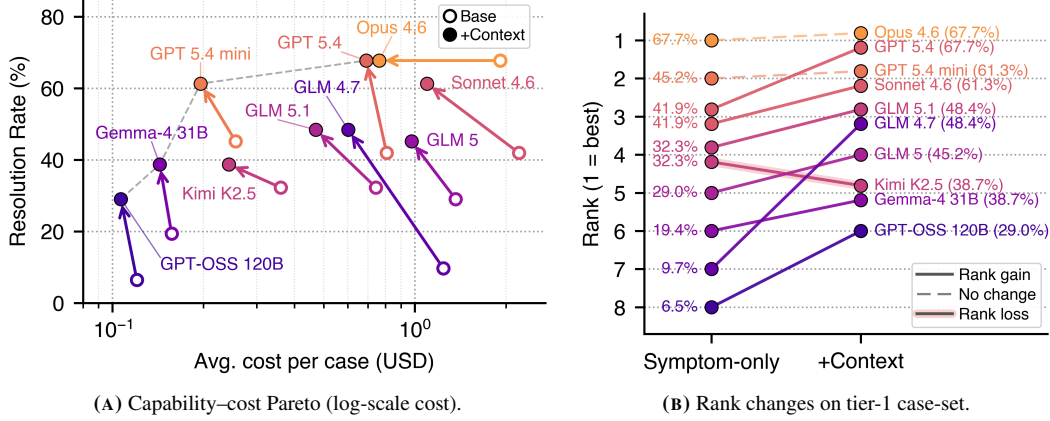


FIGURE 6: Debug-context reshapes both the capability-cost Pareto and the capability ranking on tier-1. **(a)** Each arrow traces a model from symptom-only (hollow) to context-augmented (solid). Weaker models’ arrows point up (capability gain) and stronger models’ arrows point left (cost reduction at unchanged pass rate). **(b)** Solid lines denote rank gains, dashed lines no change, and lines with a soft red halo denote losses; ties share a rank with vertical jitter. The presence of debug-context reshuffles the leaderboard.

bench Verified, but span 58 pp (9.7%–67.7%) on DDBENCH tier-1 case-set. The wide spread is not an artifact of including weak models. Specifically, Claude Opus 4.6 and GPT 5.4 are 25.8 pp apart on DDBENCH despite being only 3.8 pp apart on SWE-bench Verified.

We quantify this with a pairwise bootstrap test over the $\binom{6}{2} = 15$ model pairs (the full matrix of results can be found in Appendix §G.1). Significance concentrates on pairings involving Opus 4.6 (5 pairs) or GLM 4.7 (4 additional pairs). The two pairs SWE-bench Verified resolves are both contained in DDBENCH’s nine and involve GLM 4.7, the model with the lowest single-process score; the remaining seven DDBENCH-resolved pairs cluster tightly on SWE-bench Verified yet separate sharply once cross-process reasoning is required, exposing a capability axis the single-process benchmark does not measure.

DDBENCH shows that “frontier model” is not a homogeneous category on the cross-process reasoning axis. Concretely, models clustered tightly on single-process code repair benchmarks diverge substantially on distributed debugging tasks that require reasoning about cross-process causality, non-deterministic interleavings, and protocol invariants.

3.3 Effect of Debug-Context on Agent Performance

Across all 10 models on tier-1, the context-augmented condition raised the aggregate pass-rate from 32.6% to 50.6% (+18.1 pp). The lift is broad and uniform in direction: nine of ten models improved, one tied (Claude Opus 4.6, already at 67.7%), and none regressed. Across the 310 paired runs, the debug-context flipped 71 cases from FAIL to PASS and only 15 from PASS to FAIL, with the remaining 224 unchanged, a 4.7:1 ratio of gains to regressions (Appendix §G.5).

Pass-rate gains and efficiency gains divide along model baseline capability. The weakest models primarily gain pass-rate, the strongest mainly gain efficiency, and mid-range models gain both. More specifically, GLM 4.7 rose from 9.7% to 48.4% (+38.7 pp) and GPT OSS from 6.5% to 29.0% (+22.6 pp), the largest absolute pass-rate gains in the cohort. Claude Opus 4.6, by contrast, showed zero pass-rate improvement; its token consumption dropped from 2.46M to 0.75M (−69%) and its steps-to-completion dropped from 63.6 to 37.8 (−41%) (§3.4). Mid-range models gained 16–19 pp in pass-rate, with GPT 5.4 the notable exception, gaining +25.8 pp from a 41.9% baseline to 67.7%. Per-model breakdowns appear in Table 1. This asymmetry tracks how models cope with missing context. Strong models have the capacity to compensate by exploring more aggressively and reasoning longer; adding context removes that exploration overhead while leaving pass rate unchanged. Weak models lack that capacity, so a narrowed search space converts directly into solved cases.

Debug-context can reorder the leaderboard at fixed budget. A weaker model with debug-context can outperform a stronger model without it on both axes simultaneously. For instance, GLM 4.7 augmented with context reaches 48.4% pass rate at \$0.60 per case, beating Sonnet 4.6 in the symptom-only condition (41.9% at \$2.21) on both pass rate and cost. The Pareto plot in Figure 6a traces this

geometry across the cohort: each model’s arrow points along its dominant gain, capability for weak models, cost for strong models, and both for mid-range. The rank diagram in Figure 6b shows the practical consequence. Concretely, GLM 4.7 climbs from rank 7 to tied 3, and GPT 5.4 ties Opus 4.6 for first. For agent-tool developers, these findings suggest that a well-designed debug-context can lift a weaker, cheaper model close to the operating point of a stronger, more expensive one, making tool design a competitive lever against model upgrades.

Effect by bug family and the abstraction-distance principle.

Table 2 breaks down the effect by bug category on tier-1, revealing how the lift tracks the size of the diagnostic search space that debug-context helps to narrow. Concurrency hazards category has the biggest pass-rate lift (+42.5 pp), showing that interleavings observable in stack traces and race-detector reports can substantially reduce the search space. The replication and consistency violation category (state divergence visible across replica logs) gain +18.9 pp. However, the protocol and recovery failure category gain only +11.8 pp, suggesting the protocol-level failures might require a more sophisticated approach to curate a more effective debug-context.

TABLE 2: Pass-rate lift by bug category on tier-1, pooled across 10 models. “Input-triggered crashes” is omitted because its cases reside exclusively in tier-3.

Bug Category	Sym.	+Ctx.	Δ (pp)
Concurrency hazards (n=4)	32.5%	75.0%	+42.5
Replication & consistency (n=9)	38.9%	57.8%	+18.9
Protocol & recovery (n=13)	27.7%	39.5%	+11.8
Resource & delivery (n=5)	34.0%	48.0%	+14.0

When it misleads? Despite the general effectiveness of debug-context, the per-case outcome reveals that it can also mislead the agent. We performed a case study on three tier-1 cases (ZooKeeper-4846, Raft-268, Dragonfly-6687) in Appendix §G.6 and reached one finding: when the abstraction distance between observation (supplied as prompts or debug-context bundle) and problematic code location (the location where the actual bug resides) is small, debug-context acts as a search-space reducer; however, when the gap is large, the additional debug-context anchors the agent to the wrong code locations in the first place and can mislead even strong models. For instance, a test case can faithfully reproduce a bug, but if the failure observation in the test case is a high-level symptom and the root cause is a low-level protocol violation, then the observation via test cases is far from actual cause and can mislead the agent about where to look. This abstraction distance is the primary axis along which debug-context for LLM-based debugging agents should be designed.

Statistical significance. Per-model paired bootstraps (10,000 resamples) yield positive debug-context effects for 9 of 10 models, with 6 clearing $p < 0.05$ individually at $n = 31$ (Sonnet 4.6, GPT 5.4, GLM 5.1, GLM 4.7, Gemma-4 31B, GPT-OSS 120B). The remaining three models show positive point estimates whose bootstrap CIs cross zero, as expected given limited per-model power. The mixed-effects logistic regression (introduced early in §3.1) on all 620 outcomes gives $\hat{\beta}_{\text{ctx}} = +1.38$ (SE 0.16, 95% CI [+1.07, +1.68]; OR 3.97; $z = 8.84$, $p < 10^{-17}$), with an average marginal effect of $\Delta = +18.0$ pp matching the raw +18.1 pp lift. The variance components carry two further messages. First, case difficulty dominates model identity as a source of outcome variation ($\hat{\sigma}_{\text{case}}^2 = 4.73$ vs. $\hat{\sigma}_{\text{model}}^2 = 1.21$ on the log-odds scale), indicating DDBENCH’s signal comes from genuine per-case difficulty, not from noisy model rankings. Second, $\hat{\beta}_{\text{ctx}}$ is unchanged when these two random intercepts are included, so debug-context helps across the full difficulty distribution rather than only on easy cases.

3.4 Debug-Context as a Diagnostic Accelerator

Beyond pass rate, debug-context reshapes how agents debug (Table 3): per-case steps, first-edit time, tokens, and exploratory reads drop by 25–45%, while edit count is unchanged. We also observe that the narrowing is over the diagnostic search space rather than over the fix itself, and the savings persist regardless of outcome. Concretely, our analysis shows 12.4 fewer steps on concordant-PASS pairs (86 cases), 7.2 on concordant-FAIL (138), and 24.2 on FAIL→PASS flips (71). Claude Opus 4.6, the most expensive model evaluated, dropped from 2.46M to 0.75M tokens per case (−69%) at un-

TABLE 3: Behavioral metrics across 310 paired runs on tier-1. Debug-context reduces exploration effort consistently while edit count remains stable.

Metric	Sym.	+Ctx.	Δ
Mean steps	56.7	42.7	−13.9
Mean first-edit step	32.2	20.5	−11.7
Mean tokens	1.69M	0.93M	−0.75M
Mean read actions	23.9	17.3	−6.6
Mean explore actions	10.7	6.7	−3.9
Mean edit count	3.0	3.2	+0.1

changed pass rate. Across the cohort, debug-context roughly halved the mean token consumption without loss in diagnostic quality. The agent reaches the relevant code region faster and wastes less effort on dead ends. Overall, debug-context acts simultaneously as a capability booster for weaker models and a cost reducer for stronger ones.

3.5 Debug-Context-Enabled Performance Lift Scales with Diagnostic Search-Space Size

We also evaluated on tier-2 (15 cases) and tier-3 (14 cases) to test whether the tier-1 effect is structural to distributed debugging or an artifact of selecting bugs LLMs cannot solve unaided (Figure 7).

Debug-context lifted aggregated mean pass rates on both tiers: 48.7% $\rightarrow$ 63.3% on tier-2 (+14.7 pp) and 70% $\rightarrow$ 77.1% on tier-3 (+7.1 pp). On tier-2, the remaining 2 tied (Claude Sonnet 4.6 and GPT 5.4 mini, both already at 66.7%), none regressed, and debug-context flipped 21 cases FAIL $\rightarrow$ PASS against only 4 in reverse. On tier-3, 1 tied and 1 showed a minor regression (GLM 5.1, -7.1% , within binomial noise at $n = 14$); however, we note that the symptom-only baseline (70%) is already more than twice that of tier-1 (32.6%), since tier-3 cases are predominantly single-process input-triggered crashes that models can often diagnose from the symptom alone.

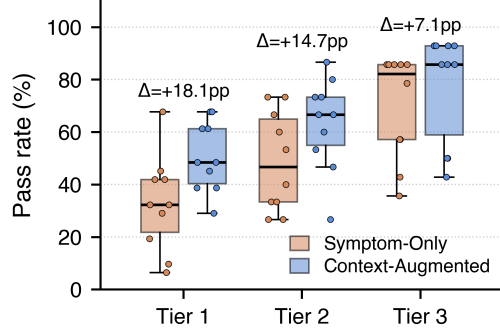


FIGURE 7: Pass rates across tiers 1–3: debug-context lift on pass-rate attenuates as the diagnostic search space shrinks. Dots show per-model pass rates.

Debug-context value tracks diagnostic search-space size. The pass-rate lift narrows monotonically across tiers, yet remains obvious even in the case where the symptom-only baseline exceeds 70%. Tier difficulty in DDBENCH is, by construction, a proxy for search-space size: tier-3 localizes to a single process and a triggering input; tier-2 spans multi-process state; tier-1 requires reasoning over distributed protocol state and concurrent interleavings. The tier-2 and tier-3 results confirm that debug-context produces a measurable lift but the lift shrinks as the search space narrows.

4 Limitations and Future Work

Single agent scaffold. We only used mini-swe-agent [30] as the agent scaffold. Nevertheless, DDBENCH is designed to be agent-agnostic so that any agent that can read a repository, edit files, and signal completion can be plugged in via the addon mechanism described in Appendix §E, and the released codebase already ships working addons for Claude Code [2] and Codex [3]. We have not yet run a controlled cross-agent-scaffold replication of the symptom-only vs. context-augmented contrast because of compute constraints. A multi-scaffold evaluation isolating the contribution of agent design from model capability is a natural next step.

No content-level ablation of the debug-context bundle. The two-condition contrast (symptom-only vs. context-augmented) treats the bundle as an atomic unit and so cannot tell us *which* component of the bundle (logs, traces, goroutine dumps, code-investigation notes) drives the gain, or how the gain scales with the number and type of files supplied. The debug-context statistics in §2.3 make a content-level ablation tractable on the existing infrastructure, and a follow-up study holding the model fixed while varying the debug-context subset would attribute the headline lift to specific signal types.

Single curated release. DDBENCH’s 60 cases reflect the multi-stage quality bar described in §2 (semi-agentic mining, two-direction oracle verification, three-pass sanitization, five-researcher manual audit) rather than a scale ceiling. The agent-assisted mining protocol (Appendix §C) is released alongside the benchmark so that the case set can grow under the same contract, and we treat scale-up across additional systems and bug families as ongoing work.

5 Conclusion

Distributed-system debugging is the regime where two distinct questions about coding agents become cleanly measurable: which models reason better on cross-process interactions and nondeterministic timings, and how much additional debugging information (debug-context) changes agent behavior.

DDBENCH operationalizes this regime as a controlled instrument with 60 historical bugs evaluated under matched symptom-only and context-augmented conditions, surfacing a 61 pp cross-model spread on tier-1 and an asymmetric context lift that reorders rankings at fixed budget. DDBENCH thereby points to two frontiers: model reasoning across processes and non-deterministic schedules, and tooling that supplies well-curated debugging context to lift performance and reduce cost simultaneously.

References

- [1] SWE-agent/mini-swe-agent. SWE-agent, April 2026. URL <https://github.com/SWE-agent/mini-swe-agent>.
- [2] Claude Code, 2026. URL <https://code.claude.com/docs/en/overview>. Accessed: 2026-04-19.
- [3] Codex CLI, 2026. URL <https://developers.openai.com/codex/cli>. Accessed: 2026-04-19.
- [4] Toufique Ahmed, Supriyo Ghosh, Chetan Bansal, Thomas Zimmermann, Xuchao Zhang, and Saravan Rajmohan. Recommending root-cause and mitigation steps for cloud incidents using large language models. In *Proceedings of the 45th International Conference on Software Engineering, ICSE '23*, pages 1737–1749, Melbourne, Victoria, Australia, July 2023. IEEE Press. ISBN 978-1-6654-5701-9. doi: 10.1109/ICSE48619.2023.00149. URL <https://dl.acm.org/doi/10.1109/ICSE48619.2023.00149>.
- [5] Ibragim Badertdinov, Alexander Golubev, Maksim Nekrashevich, Anton Shevtsov, Simon Karasik, Andrei Andriushchenko, Maria Trofimova, Daria Litvintseva, and Boris Yangel. SWE-rebench: An Automated Pipeline for Task Collection and Decontaminated Evaluation of Software Engineering Agents. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, October 2025. URL <https://openreview.net/forum?id=nMpJoVmRy1>.
- [6] Dong Chen, Shaoxin Lin, Muhan Zeng, Daoguang Zan, Jian-Gang Wang, Anton Cheshkov, Jun Sun, Hao Yu, Guoliang Dong, Artem Aliev, Jie Wang, Xiao Cheng, Guangtai Liang, Yuchi Ma, Pan Bian, Tao Xie, and Qianxiang Wang. CodeR: Issue Resolving with Multi-Agent and Task Graphs, June 2024. URL <http://arxiv.org/abs/2406.01304>.
- [7] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching Large Language Models to Self-Debug. In *The Twelfth International Conference on Learning Representations*, October 2023. URL <https://openreview.net/forum?id=KuPixIqPiq>.
- [8] Yinfang Chen, Manish Shetty, Gagan Somashekar, Minghua Ma, Yogesh Simmhan, Jonathan Mace, Chetan Bansal, Rujia Wang, and Saravan Rajmohan. AIOpsLab: A holistic framework to evaluate AI agents for enabling autonomous clouds. In *arXiv.Org*, January 2025. URL <https://arxiv.org/abs/2501.06706v1>.
- [9] Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, Karmini Sampath, Maya Krishnan, Srivatsa Kundurthy, Sean Hendryx, Zifan Wang, Vijay Bharadwaj, Jeff Holm, Raja Aluri, Chen Bo Calvin Zhang, Noah Jacobson, Bing Liu, and Brad Kenstler. SWE-bench pro: Can AI agents solve long-horizon software engineering tasks?, November 2025. URL <http://arxiv.org/abs/2509.16941>.
- [10] Zhiyu Fan, Kirill Vasilevski, Dayi Lin, Boyuan Chen, Yihao Chen, Zhiqing Zhong, Jie M. Zhang, Pinjia He, and Ahmed E. Hassan. SWE-Effi: Re-Evaluating Software AI Agent System Effectiveness Under Resource Constraints, September 2025. URL <http://arxiv.org/abs/2509.09853>.
- [11] Haryadi S. Gunawi, Thanh Do, Pallavi Joshi, Peter Alvaro, Joseph M. Hellerstein, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Koushik Sen, and Dhruba Borthakur. {FATE} and {DESTINI}: A framework for cloud recovery testing. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*, 2011. URL <https://www.usenix.org/conference/nsdi11/fate-and-destini-framework-cloud-recovery-testing>.
- [12] Zhihan Jiang, Jinyang Liu, Junjie Huang, Yichen Li, Yintong Huo, Jiazhen Gu, Zhuangbin Chen, Jieming Zhu, and Michael R. Lyu. A large-scale evaluation for log parsing techniques: How far are we?, March 2024. URL <http://arxiv.org/abs/2308.10828>.
- [13] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R. Narasimhan. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*, October 2023. URL <https://openreview.net/forum?id=VTF8yNQm66>.

- [14] Haolin Jin, Linghan Huang, Haipeng Cai, Jun Yan, Bo Li, and Huaming Chen. From LLMs to LLM-based agents for software engineering: A survey of current, challenges and future, April 2025. URL <http://arxiv.org/abs/2408.02479>.
- [15] Tanakorn Leesatapornwongsa, Jeffrey F. Lukman, Shan Lu, and Haryadi S. Gunawi. TaxDC: A Taxonomy of Non-Deterministic Concurrency Bugs in Datacenter Distributed Systems. *SIGPLAN Not.*, 51(4):517–530, March 2016. ISSN 0362-1340. doi: 10.1145/2954679.2872374. URL <https://dl.acm.org/doi/10.1145/2954679.2872374>.
- [16] Yuhe Liu, Changhua Pei, Longlong Xu, Bohan Chen, Mingze Sun, Zhirui Zhang, Yongqian Sun, Shenglin Zhang, Kun Wang, Haiming Zhang, Jianhui Li, Gaogang Xie, Xidao Wen, Xiaohui Nie, Minghua Ma, and Dan Pei. OpsEval: A comprehensive benchmark suite for evaluating large language models’ capability in IT operations domain. In *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering, FSE Companion ’25*, pages 503–513, New York, NY, USA, July 2025. Association for Computing Machinery. ISBN 979-8-4007-1276-0. doi: 10.1145/3696630.3728572. URL <https://dl.acm.org/doi/10.1145/3696630.3728572>.
- [17] Mike A. Merrill, Alexander Glenn Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E. Kelly Buchanan, Junhong Shen, Guanghao Ye, Haowei Lin, Jason Poulos, Maoyu Wang, Marianna Nezhurina, Di Lu, Orfeas Menis Mastromichalakis, Zhiwei Xu, Zizhao Chen, Yue Liu, Robert Zhang, Leon Liangyu Chen, Anurag Kashyap, Jan-Lucas Uslu, Jeffrey Li, Jianbo Wu, Minghao Yan, Song Bian, Vedang Sharma, Ke Sun, Steven Dillmann, Akshay Anand, Andrew Lanpouthakoun, Bardia Koopah, Changran Hu, Etash Kumar Guha, Gabriel H. S. Dreiman, Jiacheng Zhu, Karl Krauth, Li Zhong, Niklas Muennighoff, Robert Kwesi Amanfu, Shangyin Tan, Shreyas Pimpalgaonkar, Tushar Aggarwal, Xiangning Lin, Xin Lan, Xuandong Zhao, Yiqing Liang, Yuanli Wang, Zilong Wang, Changzhi Zhou, David Heineman, Hange Liu, Harsh Trivedi, John Yang, Junhong Lin, Manish Shetty, Michael Yang, Nabil Omi, Negin Raoof, Shanda Li, Terry Yue Zhuo, Wuwei Lin, Yiwei Dai, Yuxin Wang, Wenhao Chai, Shang Zhou, Dariush Wahdany, Ziyu She, Jiaming Hu, Zhikang Dong, Yuxuan Zhu, Sasha Cui, Ahson Saiyed, Arinbjörn Kolbeinsson, Christopher Michael Rytting, Ryan Marten, Yixin Wang, Jenia Jitsev, Alex Dimakis, Andy Konwinski, and Ludwig Schmidt. Terminal-Bench: Benchmarking Agents on Hard, Realistic Tasks in Command Line Interfaces. In *The Fourteenth International Conference on Learning Representations*, October 2025. URL https://openreview.net/forum?id=a7Qa4CcHak&referrer=%5Bthe+profile+of+Alex+Dimakis%5D%28%2Fprofile%3Fid%3D%7EAlex_Dimakis1%29.
- [18] Niels Mündler, Mark Niklas Mueller, Jingxuan He, and Martin Vechev. SWT-Bench: Testing and Validating Real-World Bug-Fixes with Code Agents. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, November 2024. URL <https://openreview.net/forum?id=9Y8zU011EQ>.
- [19] Shuang Sun, Huatong Song, Lisheng Huang, Jinhao Jiang, Ran Le, Zhihao Lv, Zongchao Chen, Yiwen Hu, Wenyang Luo, Wayne Xin Zhao, Yang Song, Hongteng Xu, Tao Zhang, and Ji-Rong Wen. SWE-world: Building software engineering agents in docker-free environments, February 2026. URL <http://arxiv.org/abs/2602.03419>.
- [20] Manan Suri, Xiangci Li, Mehdi Shojai, Songyang Han, Chao-Chun Hsu, Shweta Garg, Aniket Anand Deshmukh, and Varun Kumar. CodeScout: Contextual Problem Statement Enhancement for Software Agents, April 2026. URL <http://arxiv.org/abs/2603.05744>.
- [21] Minh V. T. Thai, Tue Le, Dung Nguyen Manh, Huy Phan Nhat, and Nghi D. Q. Bui. SWE-EVO: Benchmarking Coding Agents in Long-Horizon Software Evolution Scenarios, April 2026. URL <http://arxiv.org/abs/2512.18470>.
- [22] Runchu Tian, Yining Ye, Yujia Qin, Xin Cong, Yankai Lin, Yinxu Pan, Yesai Wu, Hui Haotian, Liu Weichuan, Zhiyuan Liu, and Maosong Sun. DebugBench: Evaluating debugging capability of large language models. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar, editors, *Findings of the Association for Computational Linguistics: ACL 2024*, pages 4173–4198, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.findings-acl.247. URL <https://aclanthology.org/2024.findings-acl.247/>.

- [23] Dong Wang, Wensheng Dou, Yu Gao, Chenao Wu, Jun Wei, and Tao Huang. Model Checking Guided Testing for Distributed Systems. In *Proceedings of the Eighteenth European Conference on Computer Systems*, pages 127–143, Rome Italy, May 2023. ACM. ISBN 978-1-4503-9487-1. doi: 10.1145/3552326.3587442. URL <https://dl.acm.org/doi/10.1145/3552326.3587442>.
- [24] 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, April 2025. URL <http://arxiv.org/abs/2407.16741>.
- [25] Yilun Wang, Guangba Yu, Haiyu Huang, Zirui Wang, Yujie Huang, Pengfei Chen, and Michael R. Lyu. Cloud-OpsBench: A Reproducible Benchmark for Agentic Root Cause Analysis in Cloud Systems, February 2026. URL <http://arxiv.org/abs/2603.00468>.
- [26] Yuxiang Wei, Olivier Duchenne, Jade Copet, Quentin Carbonneaux, Lingming Zhang, Daniel Fried, Gabriel Synnaeve, Rishabh Singh, and Sida I. Wang. SWE-RL: Advancing LLM reasoning via reinforcement learning on open software evolution, December 2025. URL <http://arxiv.org/abs/2502.18449>.
- [27] Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Agentless: Demystifying LLM-based Software Engineering Agents, October 2024. URL <http://arxiv.org/abs/2407.01489>.
- [28] Chunqiu Steven Xia, Zhe Wang, Yan Yang, Yuxiang Wei, and Lingming Zhang. Live-SWE-agent: Can Software Engineering Agents Self-Evolve on the Fly?, November 2025. URL <http://arxiv.org/abs/2511.13646>.
- [29] Chengxing Xie, Bowen Li, Chang Gao, He Du, Wai Lam, Difan Zou, and Kai Chen. SWE-fixer: Training open-source LLMs for effective and efficient GitHub issue resolution. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar, editors, *Findings of the Association for Computational Linguistics: ACL 2025*, pages 1123–1139, Vienna, Austria, July 2025. Association for Computational Linguistics. ISBN 979-8-89176-256-5. doi: 10.18653/v1/2025.findings-acl.62. URL <https://aclanthology.org/2025.findings-acl.62/>.
- [30] 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 *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, November 2024. URL [https://openreview.net/forum?id=mXpq6ut8J3&referrer=%5Bthe%20profile%20of%20Shunyu%20Yao%5D\(%2Fprofile%3Fid%3D%20Shunyu_Yao1\)](https://openreview.net/forum?id=mXpq6ut8J3&referrer=%5Bthe%20profile%20of%20Shunyu%20Yao%5D(%2Fprofile%3Fid%3D%20Shunyu_Yao1)).
- [31] John Yang, Kilian Lieret, Carlos E. Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. SWE-smith: Scaling data for software engineering agents. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, October 2025. URL https://openreview.net/forum?id=63iVrXc8cC&referrer=%5Bthe+profile+of+Carlos+E.+Jimenez%5D%28%2Fprofile%3Fid%3D%7ECarlos_E._Jimenez1%29.
- [32] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. MODIST: Transparent Model Checking of Unmodified Distributed Systems. pages 213–228, 2009.
- [33] Guangba Yu, Pengfei Chen, Yufeng Li, Hongyang Chen, Xiaoyun Li, and Zibin Zheng. NeZha: Interpretable fine-grained root causes analysis for microservices on multi-modal observability data. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2023, pages 553–565, New York, NY, USA, November 2023. Association for Computing Machinery. ISBN 979-8-4007-0327-0. doi: 10.1145/3611643.3616249. URL <https://dl.acm.org/doi/10.1145/3611643.3616249>.

- [34] Danlong Yuan, Wei Wu, Zhengren Wang, Xueliang Zhao, Huishuai Zhang, and Dongyan Zhao. SWE-MiniSandbox: Container-free reinforcement learning for building software engineering agents, March 2026. URL <http://arxiv.org/abs/2602.11210>.
- [35] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U. Jain, and Michael Stumm. Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 249–265, 2014. ISBN 978-1-931971-16-4. URL <https://www.usenix.org/conference/osdi14/technical-sessions/presentation/yuan>.
- [36] Yue Yuan, Wenchang Shi, Bin Liang, and Bo Qin. An approach to cloud execution failure diagnosis based on exception logs in OpenStack. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pages 124–131, July 2019. doi: 10.1109/CLOUD.2019.00031. URL <https://ieeexplore.ieee.org/document/8814553>.
- [37] Daoguang Zan, Zhirong Huang, Wei Liu, Hanwu Chen, Shulin Xin, Linhao Zhang, Qi Liu, Aoyan Li, Lu Chen, Xiaojian Zhong, Siyao Liu, Yongsheng Xiao, Liangqiang Chen, Yuyu Zhang, Jing Su, Tianyu Liu, Rui Long, Ming Ding, and Liang Xiang. Multi-SWE-bench: A multilingual benchmark for issue resolving. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, October 2025. URL https://openreview.net/forum?id=MhBZzkz4h9&referrer=%5Bthe+profile+of+Zhirong+Huang%5D%28%2Fprofile%3Fid%3D%7EZhirong_Huang3%29.
- [38] Baolei Zhang, Haoran Xin, Minghong Fang, Zhuqing Liu, Biao Yi, Tong Li, and Zheli Liu. Traceback of poisoning attacks to retrieval-augmented generation. In *Proceedings of the ACM on Web Conference 2025, WWW '25*, pages 2085–2097, New York, NY, USA, April 2025. Association for Computing Machinery. ISBN 979-8-4007-1274-6. doi: 10.1145/3696410.3714756. URL <https://dl.acm.org/doi/10.1145/3696410.3714756>.
- [39] Li Zhong, Zilong Wang, and Jingbo Shang. Debug like a human: A large language model debugger via verifying runtime execution step by step. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar, editors, *Findings of the Association for Computational Linguistics: ACL 2024*, pages 851–870, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.findings-acl.49. URL <https://aclanthology.org/2024.findings-acl.49/>.
- [40] Jingyu Zhou, Meng Xu, Alexander Shraer, Bala Namasivayam, Alex Miller, Evan Tschannen, Steve Atherton, Andrew J. Beamon, Rusty Sears, John Leach, Dave Rosenthal, Xin Dong, Will Wilson, Ben Collins, David Scherer, Alec Grieser, Young Liu, Alvin Moore, Bhaskar Muppana, Xiaoge Su, and Vishesh Yadav. FoundationDB: A distributed unbundled transactional key value store. In *Proceedings of the 2021 International Conference on Management of Data, SIGMOD '21*, pages 2653–2666, New York, NY, USA, June 2021. Association for Computing Machinery. ISBN 978-1-4503-8343-1. doi: 10.1145/3448016.3457559. URL <https://dl.acm.org/doi/10.1145/3448016.3457559>.
- [41] Jieming Zhu, Shilin He, Pinjia He, Jinyang Liu, and Michael R. Lyu. Loghub: A Large Collection of System Log Datasets for AI-driven Log Analytics, September 2023. URL <http://arxiv.org/abs/2008.06448>.
- [42] Terry Yue Zhuo, Vu Minh Chien, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, Simon Brunner, Chen Gong, James Hoang, Armel Randy Zebaze, Xiaoheng Hong, Wen-Ding Li, Jean Kaddour, Ming Xu, Zhihan Zhang, Prateek Yadav, Naman Jain, Alex Gu, Zhoujun Cheng, Jiawei Liu, Qian Liu, Zijian Wang, Binyuan Hui, Niklas Muennighoff, David Lo, Daniel Fried, Xiaoning Du, Harm de Vries, and Leandro Von Werra. BigCodeBench: Benchmarking code generation with diverse function calls and complex instructions. In *The Thirteenth International Conference on Learning Representations*, October 2024. URL <https://openreview.net/forum?id=YrycTj1l10>.

A Related Work

Code-repair benchmarks for LLM agents. SWE-bench and its successors established the dominant template for evaluating LLM coding agents: mine historical issue–PR pairs from open-source repositories, package each as a faulty worktree plus a hidden test oracle, and score agents by whether their patch passes [13, 37, 21, 22, 9, 26, 5, 31, 42, 18]. These benchmarks have been invaluable, but they inherit two structural limits from their source material. First, they draw almost exclusively from single-process Python (and, in the multilingual extensions [37], single-process Java/TypeScript/Go) repositories, where the reasoning challenge is primarily cross-file localization within one address space. Second, the oracle is a standard unit or integration test that can be run deterministically in a sandbox, which excludes bugs whose reproduction requires multi-node clusters, non-deterministic scheduling, or runtime-state capture. DDBENCH inherits the faulty-worktree-with-hidden-oracle contract but relaxes both limits. Cases are drawn from distributed systems in five compiled and JVM languages, and the oracle is a containerized harness (oftentimes end-to-end integration tests) rather than a single test invocation, which enables us to test concurrency hazards and protocol-level bugs that unit-test oracles often structurally cannot express.

IT-operations and cloud-incident benchmarks. A parallel line of work evaluates LLM agents on diagnosis of distributed and cloud systems deployment from telemetry, utilizing logs, metrics, traces, and alerts [8, 25, 16, 4]. However, these benchmarks target only operational diagnosis of deployment issues rather than the repair of actual distributed systems. These benchmarks establish that cross-service reasoning is a distinct capability axis, and many inject controlled faults (latency, CPU, packet loss, crash) into microservice deployments to generate incidents on demand. Their scoring, however, targets root-cause identification or remediation action selection rather than producing a code patch. The agent’s output is a natural-language attribution or a pre-defined mitigation, not a source *diff* verified by a reproduce oracle. DDBENCH is orthogonal and complementary. DDBENCH shares the cross-process reasoning load these benchmarks exercise but closes the loop on source-level repair, so the ground truth is an executable oracle rather than a labeled root cause. This combination is what lets us measure the effect of debugging context on agent performance: an agent can consume runtime signals and gets scored on whether it produces a correct patch.

LLM agents for debugging and program repair. A growing body of work studies LLMs and agent scaffolds for bug localization, fault repair, and test-time reasoning over code [24, 30, 27, 6, 29]. Most of this work is developed and tuned against single-process repair benchmarks. Under this paradigm, the dominant agent patterns are shaped as read–edit–test loops with localized retrieval and repository-wide code search. A separate strand augments LLM debugging with runtime signals. Namely, executing programs to collect stack traces, observing variable state, or consulting coverage and profiling tools [39, 7, 38]. DDBENCH extends this line into the distributed setting, where the relevant runtime signals (cluster logs, goroutine dumps, race-detector reports, replica state) are qualitatively different and are not consistently recoverable by re-executing a single process. Our context-augmented condition makes this information channel a controlled input variable rather than an agent-internal implementation detail, which lets future agent and tool work be evaluated on the axis of *which runtime signals it surfaces*, not only on end-to-end pass rate.

Distributed-systems bug studies and testing. Empirical studies of real-world distributed-systems failures have repeatedly shown that the hardest bugs arise from protocol-level invariants, cross-node concurrency, and partial failures rather than from within-process logic errors, and have motivated a long line of specialized testing and model-checking infrastructure in the distributed-system community [15, 35, 40, 11, 32, 23]. This line of work provides both the taxonomy of failure modes we target (protocol/recovery, replication/consistency, concurrency hazards, input-triggered crashes, resource/delivery) and the evidence that such bugs cannot be reliably caught by single-process testing. Our contribution is not a new testing technique but a benchmark that imports the bug distribution and reproducibility standards from this literature into the LLM-agent evaluation setting. Cases are drawn from the classes these studies identify as hardest; each is paired with a hardened reproducer; and the oracle accepts any patch that makes the reproducer pass rather than requiring a textual match (our manual examinations ensure this).

Datasets for runtime signals. A separate line of work has released large-scale datasets of real system logs, traces, and telemetry collected from production and benchmark distributed deployments, originally built for log parsing, anomaly detection, and trace-based root-cause analysis (RCA) [41,

12, 36, 33]. These datasets are valuable runtime-signal sources, but they are not directly drop-in replacements for DDBENCH’s evaluation harness, since they are not paired with a faulty code commit, a sanitized symptom description, or a reproduce oracle; an agent consuming them therefore cannot be scored on whether its resulting patch compiles and passes. DDBENCH is designed so that such datasets can be brought in through its debug-context channel. Each case defines a bounded slot (the per-case evidence folder) that the harness mounts into the agent’s container, and the content of that slot is deliberately decoupled from the oracle and the symptom file. Future work can therefore treat these public log/trace corpora, fault-injection telemetry, and incident postmortems as alternative debug-context sources. Researchers can select the subset whose system or failure mode matches a DDBENCH case, transform it into the debug-context slot’s format, and re-evaluate agents under the same symptom-only vs. context-augmented contrast we use in §3. This turns the existing log/trace dataset ecosystem into a first-class axis of ablation for distributed-systems agent debugging evaluation.

B Dataset Details

B.1 Per-Project Coverage Breakdown

TABLE 4: Per-project coverage of DDBENCH. DDBENCH spans 13 open-source distributed systems across five languages and six system families, partitioned into three difficulty tiers (tier-1, tier-2, tier-3).

Project	Language	System family	Number of Cases	Tier-1	Tier-2	Tier-3
Dragonfly	C++	kv-store	14	1	1	12
NATS	Go	messaging	9	5	3	1
Kvrocks	C++	kv-store	8	4	3	1
etcd	Go	kv-store	7	5	2	0
hashicorp/raft	Go	consensus	6	4	2	0
Consul	Go	service-mesh	3	2	1	0
Kafka	Java	streaming	3	3	0	0
ZooKeeper	Java	consensus	3	3	0	0
BookKeeper	Java	consensus	2	2	0	0
raft-rs	Rust	consensus	2	0	2	0
Ra	Erlang	consensus	1	1	0	0
RabbitMQ	Erlang	messaging	1	1	0	0
RocksDB	C++	storage-engine	1	0	0	1
Total	5 langs	6 families	60	31	15	14

Table 4 reports the full per-project breakdown of DDBENCH’s 60 cases across the 13 projects we mine from, annotated with each project’s main implementation language, system family, and per-tier case count. Three observations follow from the table. First, case count per project is skewed. Dragonfly contributes 14 cases while Ra, RabbitMQ, and RocksDB each contribute only 1, which reflects how active each project’s issue tracker is and how often its confirmed bugs satisfy our cross process reasoning criterion rather than any sampling quota on our side. Second, tier composition varies sharply by system family. Consensus libraries (etcd, hashicorp/raft, Kafka, ZooKeeper, BookKeeper, Ra) place nearly every case in tier-1 because correctness in these systems is defined by multi-node protocol invariants, whereas Dragonfly contributes 12 of its 14 cases to tier-3 because many of its confirmed bugs localize within a single process even though the deployment is distributed. Third, programming language coverage follows the production distribution of distributed infrastructure rather than a uniform sample. Go (25 cases) and C++ (23 cases) dominate, Java contributes 8, and Erlang and Rust each contribute 2 in this dataset release.

B.2 Licenses

Table 5 lists the license and upstream repository URL for every project in DDBENCH. All 13 projects are released under permissive terms (Apache 2.0, MPL 2.0, or BSL 1.1), which allows us to package each case as a pinned faulty worktree plus a containerized reproduce oracle and redistribute the bundle for benchmark use. Two projects (Ra and RabbitMQ) are dual-licensed under Apache 2.0 and MPL 2.0, and RocksDB is dual-licensed under Apache 2.0 and GPL 2.0; in each case we rely on the Apache 2.0 grant for redistribution. Dragonfly and Consul are released under BSL 1.1, which restricts commercial service offerings but permits research redistribution and benchmark use, the scope under which DDBENCH operates. No project in DDBENCH is released under a license that requires us to

TABLE 5: Per-project license and upstream repository for the 13 open-source systems in DDBENCH.

Project	License	Link
Dragonfly	BSL 1.1	https://github.com/dragonflydb/dragonfly
NATS	Apache-2.0	https://github.com/nats-io/nats-server
Kvrocks	Apache-2.0	https://github.com/apache/kvrocks
etcd	Apache-2.0	https://github.com/etcd-io/etcd
hashicorp/raft	MPL-2.0	https://github.com/hashicorp/raft
Consul	BSL 1.1	https://github.com/hashicorp/consul
Kafka	Apache-2.0	https://github.com/apache/kafka
ZooKeeper	Apache-2.0	https://github.com/apache/zookeeper
BookKeeper	Apache-2.0	https://github.com/apache/bookkeeper
raft-rs	Apache-2.0	https://github.com/tikv/raft-rs
Ra	Apache-2.0 / MPL-2.0	https://github.com/rabbitmq/ra
RabbitMQ	Apache-2.0 / MPL-2.0	https://github.com/rabbitmq/rabbitmq-server
RocksDB	Apache-2.0 / GPL-2.0	https://github.com/facebook/rocksdb

relicense derivative artifacts, and no case in the release ships modified upstream source. The faulty worktree for each case is a pinned commit of the original repository with no edits applied on top.

C Agent-Assisted Mining Methodology

C.1 Why Semi-Agentive?

We elaborate on the design rationale behind the semi-agentive pipeline introduced in §2.

Why not purely manual curation? Identifying cross-process bugs reliably requires deep familiarity with each system’s protocols and failure modes. For instance, distinguishing a genuine linearizability violation from a superficially similar client-side retry bug demands protocol-level expertise and substantial manual effort. Scaling such review across 13 systems and thousands of candidate issues demands a level of domain knowledge and expertise that is not realistic for pure manual curation.

Why not purely automated scraping? Faulty cases whose root-causing genuinely requires cross-process or interleaving-aware reasoning are a small fraction of any project’s issue tracker, and the discriminator is rather semantic. Namely, it cannot be easily expressed as a regex over titles or labels, nor reliably recovered from issue metadata alone. Agents, in contrast, can perform semantic analysis by reading discussion threads and linked PRs and applying a tightly scoped specification of what qualifies as a cross-process bug that DDBENCH demands.

Why the specific division of labor. The pipeline assigns to agents the tasks where breadth and heavy domain-expertise dominate, *e.g.*, walking issue trackers, drafting symptom files, hardening reproducers, producing debug-context bundles. This pipeline preserves for human reviewers the tasks where ground-truth judgement dominates, *e.g.*, final acceptance, quality gating, and sanitization of any upstream-fix leakage. Each agent stage operates under a narrow skill specification and is followed by either an automated verification pass (*e.g.*, oracle verification) or a human review pass.

C.2 Agentive Mining Protocol

We document the operational protocol that the agent follows during the mining, synthesis, and verification stages summarized in §2. The protocol is codified as a set of machine-readable specifications shipped with the DDBENCH release, and every agent stage in our pipeline executes against a subset of these specifications. The protocol decomposes into four phases: candidate admission, case-artifact synthesis, hermetic containerization, and pre-merge validation.

Phase 1: Candidate admission. Before any artifact is synthesized, a candidate bug must pass three gatekeeping tests that together enforce the cross-process reasoning contract. The *dual-reasoning test* rejects candidates for which a stack trace or an error log alone is sufficient for diagnosis, as well as candidates for which static code reading alone is sufficient. The sweet spot is bugs where runtime signals point toward a subsystem but code reasoning is required to understand why that subsystem fails under the observed conditions. The *fix-locus test* rejects candidates whose fix is scattered across many unrelated modules with no identifiable center of gravity, since such cases test breadth of refactoring rather than depth of diagnosis. The *issue-text test* rejects candidates whose GitHub issue

thread already names the faulty function or describes the fix strategy, since an agent could derive the patch from the issue alone and bypass the diagnostic process the benchmark is designed to measure. The agent applies these tests in sequence and records, for each admitted candidate, the fix commit hash and the buggy commit.

Phase 2: Case artifact synthesis. An admitted candidate is expanded into eight per-case artifacts plus one isolated solution document. The per-case artifacts are `case.yaml` (machine-readable metadata consumed by the orchestrator), `symptom.md` (observer-facing bug description), `reproduce.md` (human-readable reproduction steps), `verification.md` (documented correctness oracle with PASS/FAIL criteria), `setup.md` (human-readable build instructions), `setup.sh` (executable clone-and-build script with a Docker-first path and host fallback), `reproduce.sh` (executable trigger-and-verify script with the same dual-path structure), and `README.md` (entry point linking the other files). The solution document lives in a separate `cases-solution/` tree to prevent answer leakage during evaluation, and it is required to include both a *fix locus* (component name plus the one to three files changed) and a *diagnosis chain* that documents the reasoning path from symptom to root cause, explicitly annotating where runtime signals and where code reasoning each contribute. The solution file is what human reviewers read during acceptance and what the oracle-verification stage consults when resolving the case from the solution.

Phase 3: Hermetic containerization. Every case runs inside a deterministic, layered Docker image so that reproduction does not drift with the host environment. The base layer is `ubuntu:24.04`, on top of which sit language base images (`bug-bench/base-cpp`, `bug-bench/base-go`, and analogous images for Java, Erlang, and Rust) that install the pinned toolchain, debuggers, and standard client utilities such as `redis-cli`, `curl`, and `netcat`. Each project adds a thin image on top of its language base with only the build dependencies that project requires, and no source code is baked into any image. Source is instead mounted at runtime through a per-project Docker volume of the form `bug-bench-<project>-src` that holds a bare clone plus one detached worktree per case, so that cases from the same project share a single clone while remaining isolated at their respective buggy commits. Elevated capabilities are declared per case in `case.yaml` under `docker.caps` (for instance `NET_ADMIN` for cases that manipulate network namespaces) and `SYS_PTRACE` is always granted to support in-container debuggers. For environments without Docker, every `setup.sh` and `reproduce.sh` falls through to a host path that installs dependencies via `apt-get` under Ubuntu 24.04, which we retain only as a fallback since it sacrifices isolation.

Phase 4: Pre-merge validation. Before a case is admitted into the agent release package (before the human review), the protocol requires it to pass a fixed sequence of automated checks that jointly enforce oracle soundness, hermetic reproducibility, and absence of answer leakage. *Oracle soundness* is checked in both directions: `reproduce.sh` must report FAIL when run on the buggy commit and must report PASS when run on the commit produced by applying the solution document, so that the oracle has neither false positives (buggy code passing) nor false negatives (fixed code failing). *Hermetic reproducibility* is checked by running the full `setup` and `reproduce` pipeline end-to-end inside a fresh container, by running `setup` a second time to confirm idempotency (clone and build are skipped when artifacts already exist), and by running the full pipeline after a `clean` to confirm the case recovers from a clean slate. *Absence of answer leakage* is checked by cross-referencing `symptom.md` against `solution.md` for a fixed list of prohibited patterns. These patterns are:

- function or method names that appear in the fix diff;
- struct-field names or internal constants from the buggy code path;
- file paths that point to the fix location;
- descriptions of the fix strategy (e.g., “make X idempotent”);
- error strings visible only in source code rather than in user-facing output;
- PR or commit references that identify the fix.

A case that fails any check is either repaired and re-verified or rejected from the release.

Protocol artifacts. The full protocol, including the selection criteria, the case anatomy specification, the containerization rules, the authoring walkthrough, the pre-merge checklist, and the templates the agent copies from, is released with `DDBENCH` (in `mining-protocol` folder) so that future case additions and independent replications follow the same contract.

C.3 Case Containerization Verification Prompt

This prompt drives the hermetic reproducibility check of Phase 4 in the mining protocol (§C.2). The agent runs `bench.sh setup` followed by `bench.sh reproduce` across the target cases and classifies each outcome, so that the release only admits cases whose faulty worktree builds cleanly inside the pinned container and whose `reproduce.sh` correctly reports `FAIL` on the buggy commit. The prompt additionally carries a static anti-pattern scan that flags fix-specific reproducers (for instance, tests extracted at runtime from the fix commit via `FIX_COMMIT`, reflection-based assertions, or source-grep branches) so that fragile oracles can be flagged and remediated before evaluation. The prompt below is a shortened version of the skill file shipped with `DDBENCH`. We extract only the procedural core and omit implementation boilerplate, example report templates, and project-specific remediation tables.

Case Containerization Verification Prompt (shortened)

Run ``bench.sh setup`` followed by ``bench.sh reproduce`` across the specified cases and produce an aggregate
↪ containerization report.

Procedure:

1. Determine the case list (all cases under ``cases/``, or the caller supplied subset). Validate that each case directory
↪ exists and contains ``case.yaml``.
2. Pre flight oracle quality scan. For each ``reproduce.sh``, flag (but do not block) the following anti patterns that
↪ produce fragile, fix specific oracles:
 - `FIX_COMMIT` dependency (``FIX_COMMIT=`` variable)
 - ``git show`` or ``git checkout`` reaching into the fix commit
 - Reflection (``getDeclaredField``, ``getDeclaredMethod``) used as the PASS/FAIL assertion
 - Source grep used as the PASS/FAIL branch
 - ``sed -i`` targeting production source paths
3. Run verification sequentially (never in parallel, to avoid Docker resource contention). For each case:
 `SETUP_EXIT=$(./bench.sh setup <case-name>; echo $?)`
 if [\$SETUP_EXIT -eq 0]; then
 `REPRO_EXIT=$(./bench.sh reproduce <case-name>; echo $?)`
 fi
 Enforce a 30 minute setup timeout and a 5 minute reproduce timeout. Capture the last 20 lines of output for the report.
4. Classify each case into one of:
 - **VERIFIED**: setup OK and reproduce exited 1 with "FAIL:" in output (bug present on the buggy commit, as
↪ expected).
 - **FIXED**: setup OK and reproduce exited 0 with "PASS:" in output (unexpected; the buggy_commit is likely wrong).
 - **SETUP_FAIL**: setup exited non zero.
 - **REPRO_ERROR**: reproduce exited non zero without "FAIL:" in the output (script error, not bug detection).
 - **TIMEOUT**: setup or reproduce exceeded the timeout.
5. Produce a markdown report with a per case status table, a summary row count per status, and, for every non
↪ **VERIFIED** case, the last 20 lines of output plus a suggested next step (check build dependencies, verify
↪ `buggy_commit` resolves, increase timeout).
6. After the main run, list cases that passed verification but still tripped a pre flight anti pattern under a "Fragile
↪ Oracles" section, with the recommended remediation (inline tests as heredocs, behavioral Mockito verify, ship
↪ regression tests as static inject_test/ files).

Notes:

- A **VERIFIED** result only confirms the oracle fires on the buggy commit; soundness in the other direction is covered
↪ by the Oracle Verification prompt.
- Repeated setup failures on cases from the same project typically indicate a stale project Dockerfile rather than per
↪ case bugs.

C.4 Oracle Verification

This subsection details the oracle-verification step from Phase 4 of the mining protocol (§C.2): the two-direction soundness check, the iterative hardening loop applied when a case fails first verification,

the design choice not to verify against the upstream patched commit, the role of the per-case reference solution, the manual acceptance pass, and the verification prompt the agent runs.

Two-direction soundness check. For each case, the agent first runs the oracle on the buggy commit and confirms FAIL, then applies the fix documented in `solution.md` inside the container, rebuilds, and reruns the oracle to confirm PASS. A case is admitted only if the oracle is sound in both directions, carrying no false positive (buggy code passing) and no false negative (fixed code failing). A fix-specificity scan additionally distinguishes *sound* oracles from *sound-but-fragile* oracles, since a fragile oracle can pass the two-direction check while still rejecting valid alternative fixes at evaluation time (e.g., runtime-extracted tests, reflection-based assertions, source-grep branches).

Iterative hardening loop. A reproducer that passes the two-direction check on the first attempt is rare; the typical path is iterative hardening. Each iteration tightens one of three knobs: setup determinism (pinning seeds, fixed cluster sizes, blocking until quorum is reached), timing (sleep windows replaced with explicit wait-for-condition probes, stress phases lengthened to expose interleavings), and repetition (the trigger phase is run k times and the case is admitted only if it fires every time). We cap the loop at five iterations: if a candidate’s oracle has not stabilized by then, the case is dropped from the release rather than patched around. This preserves the invariant that every released oracle is reliable on its own terms, at the cost of admitting fewer cases.

Why not verify against the upstream patched commit. A natural alternative would be to define the oracle as “reproduce.sh fails on the buggy commit and passes on the upstream fix commit.” We rejected this because distributed-system fixes often land months or years after the bug was filed, and by the time the upstream patch is merged the surrounding code has frequently drifted (API renames, dependency bumps, build-system changes). An oracle pinned to the upstream patched commit then conflates an underlying-bug regression with API drift, which would inflate false negatives and reduce benchmark separability. We therefore verify against a per-case reference solution (below) applied directly on top of the buggy commit, where the patch context is stable.

Reference solution as Markdown, not gold patch. The reference solution shipped in `cases-solution/` is a human-readable Markdown document, not a checked-in code diff. Two reasons motivate this choice. First, a Markdown solution forces the curator to articulate the diagnosis chain — the artifact human reviewers read during acceptance and the artifact a future researcher needs to audit a contested case. Second, since the released oracle accepts *any* patch restoring the violated invariant rather than matching a textual fix, a single canonical diff would mislead readers into believing a textual match is required. The Markdown solution is consumed only by the oracle-verification stage (which applies the fix described in the document and reruns the oracle) and by human reviewers; it is never exposed at evaluation time.

Manual acceptance. After the automated checks above, every surviving case is reviewed by the five distributed-systems researchers who authored DDBENCH. Reviewers re-read the agent-produced `symptom.md`, `evidence/`, and `solution.md` against the original GitHub issue and PR, and perform a final round of symptom and debug-context sanitization on top of the automated leakage scan. A case is admitted only when reviewers sign off; cases that fail review are repaired and re-verified, or dropped from the release.

Verification prompt. The prompt below is a shortened version of the skill file shipped with DDBENCH. We extract the procedural core and omit the report templates and the full remediation guidance.

Oracle Verification Prompt (shortened)

Verify that a case's reproduce script is a sound correctness oracle: it must FAIL on the buggy commit AND PASS after
→ applying the documented fix.

Procedure:

1. Read case metadata. For each case, load ``cases/<case-name>/case.yaml`` and
→ ``cases-solution/<case-name>/solution.md``. Extract the buggy commit, the fix commit hash (from `solution.md`
→ "Commit" field), and the build command. If no fix commit is recorded, abort with an explicit error, since
→ soundness cannot be verified in both directions.

2. Direction 1 (buggy commit -> expect FAIL).
`./bench.sh setup <case-name>`
`./bench.sh reproduce <case-name>`
 Expected: exit code 1 with "FAIL:" in output. Record the exit code, the last 10 lines of output, and the verdict FAIL
 ↪ (correct) or PASS (unsound, a false positive).
3. Fix specificity scan. Before running Direction 2, scan ``reproduce.sh`` for anti patterns that pass the soundness check
 ↪ but reject valid alternative fixes:
 - FIX_COMMIT dependency (HIGH): test pulled from the fix commit.
 - ``git show`` or ``git checkout`` from the fix commit (HIGH): files pulled from the fix tree at run time.
 - Reflection used as the PASS/FAIL assertion (HIGH): checks code structure rather than behavior.
 - Source grep branching (MEDIUM): detects fix text instead of fix effect.
 - ``sed -i`` on production source (MEDIUM): injects accessors to read internal state.
 If a HIGH severity pattern is found, the oracle is flagged SOUND-BUT-FRAGILE in the final report even if both
 ↪ directions pass, together with a remediation pointer (inline test as a heredoc, replace reflection with a behavioral
 ↪ mock, ship regression tests as static `inject_test/` files).
4. Direction 2 (fix commit -> expect PASS). Apply the fix inside the container and rerun the oracle:
`./bench.sh shell <case-name> --cmd "`
`cd /workspace/worktrees/<case-name> &&`
`git checkout <fix-commit> &&`
`[git submodule update --init --recursive] &&`
`<rebuild-command> &&`
`bash /bench/reproduce.sh`
`"`
 Expected: exit code 0 with "PASS:" in output. Record the exit code, the last 10 lines of output, and the verdict PASS
 ↪ (correct) or FAIL (unsound, a false negative). For race or flaky oracles, repeat at least three times.
5. Reset the worktree back to the buggy commit (`./bench.sh reset <case-name>``) so the case is clean for downstream
 ↪ use.
6. Emit a per case report that states both directions, their verdicts, and the overall oracle verdict:
 - SOUND: both directions correct and no fix specific anti pattern found.
 - SOUND-BUT-FRAGILE: both directions correct but at least one HIGH severity anti pattern; the oracle works as a
 ↪ regression test but will reject valid alternative fixes.
 - UNSOUND: either direction fails. Classify as false positive (buggy code passes) or false negative (fixed code fails),
 ↪ and point at the likely cause (timing sensitivity, insufficient iterations, wrong condition, wrong fix commit).

C.5 Debug-Context Generation Prompt

This prompt drives debug-context synthesis for the context-augmented condition introduced in §2. The agent acts as a developer who has read only `symptom.md` and has entered the container at the buggy commit, gathers debugging clues using standard tools (stack traces, race-detector reports, goroutine or thread dumps, anomalous logs, test failures, and short code-reading notes), and writes them into a per-case `evidence/` folder. The prompt enforces two invariants critical to benchmark validity: (a) every piece of debug context must be obtainable with only the artifacts visible to the evaluation agent (no `reproduce.sh`, no injected regression tests, no access to `solution.md` during investigation), and (b) the debug context must remain one step removed from the root cause (it may point at an area worth investigating, but it must not name the defect, contrast buggy with correct code, or trace the full causal chain). The prompt below is a shortened version of the skill file shipped with DDBENCH. We extract the procedural core and the anti-leak checklist, and omit the per-language tool examples, placeholder templates, and file-naming table.

Evidence Generation Prompt (shortened)

You are a developer investigating a bug from inside the Docker container, starting from ``symptom.md`` and working
 ↪ only with artifacts visible to the evaluation agent at the buggy commit. Your job is to gather debugging clues
 ↪ (crash backtraces, anomalous logs, race detector reports, test failures, server behavior, code reading notes) and
 ↪ write them into ``cases/<case-name>/evidence/`` without revealing the fix.

Core principle. The container holds the full built source and standard tools (GDB, Go toolchain, Maven, cargo, rebar3, redis-cli, etcdctl, jstack, and similar). It does NOT hold ``reproduce.sh``, ``bench.sh``, injected test files, the ``cases/`` tree, or the ``cases-solution/`` tree. Every piece of evidence must be obtainable from that strictly limited view.

Two evidence shapes, a case may have both:

- Runtime evidence: raw output from a tool, with a single provenance line naming the command. No interpretation, no annotations, no decoded constants. Only the tool's own output.
- Code investigation evidence: a short "What was searched", a block of real grep/sed/cat output from inside the container (not hand transcribed), and a one to two sentence developer hypothesis. The hypothesis must stop at "this area is worth investigating further" and must NOT name the exact defect, contrast buggy with correct code, or trace the full causal chain.

Procedure:

1. Read only ``symptom.md``, ``reproduce.md``, ``verification.md``, and ``case.yaml``. Do NOT read ``solution.md`` yet.
 - Reading the solution before investigating contaminates the evidence by letting the agent reverse engineer clues from the known answer.
2. Read ``reproduce.sh`` as a guardrail only, to detect whether it injects test files. Any test file it injects is off limits for evidence, since the evaluation agent does not have that file.
3. Enter the container at the clean buggy commit via ``bench.sh setup`` then ``bench.sh shell`` (never via ``bench.sh reproduce``). Reset first if the workspace was dirtied.
4. Investigate. Pick tools appropriate to the bug category: GDB or ASAN for C++ crashes, ``go test -race`` or goroutine dumps for Go, ``jstack`` or existing JUnit tests for Java, ``rebar3 ct`` for Erlang, ``cargo test`` for Rust. Capture all output to ``/tmp/evidence-raw-<case-name>.txt`` with ``tee`` so every written line has a traceable source.
5. Write evidence files under ``cases/<case-name>/evidence/``. Runtime files carry only a provenance line and raw output. Code investigation files carry "What was searched", actual grep/sed output with surrounding context (20 to 40 lines, not a three line snippet), and a one to two sentence hypothesis. The provenance line must reference a command a developer runs inside the container (``go test``, ``gdb``, ``mvn test``, ``grep -n``), never ``bench.sh`` or host paths.
6. Anti leak checks. Only now, read ``solution.md`` and run these checks against the drafted evidence. Delete or rewrite any file that fails:
 - Agent reproducibility: could an agent inside the container actually run this exact command on the buggy commit and get this output?
 - Injected test: is this output from a test that ``reproduce.sh`` would inject? If yes, delete.
 - Native output: every line in the runtime ``## Output`` block is raw tool output, with no added commentary.
 - Fix from evidence: could a reader identify the exact function and change from this file alone? If yes, the file is a diagnosis, not a clue.
 - Curation: for runtime output, include realistic surrounding context (other goroutines in a dump, surrounding log lines) so the smoking gun is not artificially isolated.
 - Term check: the evidence does not carry variable names from the fix diff or internal constants decoded from source.
 - Solution comparison: the evidence does not restate the solution's root cause in different words.
7. Report the investigation approach, the files written, and confirm that the leak checks and the agent reproducibility check both passed.

C.6 Sanitization Prompt

This prompt implements the automated leakage pass described in Phase 4 of the mining protocol (§C.2) and complements the human sanitization step noted in §2. The agent cross-references `symptom.md` against `solution.md` along both a lexical axis (prohibited identifiers, paths, commit hashes, prior issue references) and a semantic axis (internal-mechanism disclosure, regression provenance, subsystem or code-path narrowing, inverted fix hints, and diagnostic equivalence to the full solution). Each finding is triaged into one of three *information tiers*, which are an internal sanitization construct and should not be confused with the case-difficulty tiers (tier-1/tier-2/tier-3) used elsewhere in the paper to partition cases. The information tiers classify individual facts about a bug: *observable* facts remain in `symptom.md`; *runtime-obtainable* facts are extracted into the per-case `evidence/`` slot; *source-derived* facts are redacted entirely. This tiering is the mechanism that keeps the symptom-only and context-augmented conditions cleanly separated across the benchmark. The prompt below is a shortened version of the skill file shipped with DDBENCH. We extract the tier

model and the scan procedure, and omit the debug-context-file format templates, the per-oracle-type recommendation table, and the example violation reports.

Sanitization Prompt (shortened)

Scan `cases/<case-name>/symptom.md` for information leakage (both lexical and semantic), triage each finding into
→ one of three tiers, extract tier 2 content into `evidence/`, and redact tier 3 content entirely.

Three tier information model:

- Tier 1 (keep in symptom.md): what an operator can observe by running the software. Crash messages, wrong CLI
→ output, missing data, timeouts, user facing errors.
- Tier 2 (extract to evidence/): what a developer can gather with standard runtime tools (logs, stack traces, goroutine
→ dumps, metrics, config state, protocol level captures), provided it does not reveal the fix.
- Tier 3 (redact entirely): anything that requires reading the project's source code to know. Root cause, fix location,
→ internal mechanism, regression provenance. Goes nowhere.

Fresh developer test. Every sentence in symptom.md must be writeable by someone who has only run the software and
→ observed the failure, never read the source.

Procedure:

1. Build the reference set. Load `cases-solution/<case-name>/solution.md` and extract the prohibited terms: function
→ and method names cited in "Root cause" / "Fix description" / "Diagnosis chain"; variable, struct field, and
→ constant names; file paths and component name from "Fix locus"; PR numbers and commit hashes; prior issue or
→ PR IDs cited as regression sources.
2. Lexical scan of `symptom.md`. Grep for each prohibited term, for file path fragments, internal constants and magic
→ numbers, and for PR, commit, or prior issue references.
3. Semantic scan of `symptom.md`. For each sentence, check these leakage shapes:
 - Internal mechanism disclosure: sentences that describe how the bug works internally rather than what the user sees.
 - Regression provenance: references to a specific prior change, issue, PR, or commit that introduced the bug.
 - Subsystem or code path narrowing: naming internal protocol phases, subsystems, or code paths not visible in user
→ facing output.
 - Fix direction hints: sentences whose inverse is effectively the fix ("does not populate X", "does not call Y after Z").
 - Diagnostic equivalence: the symptom file, taken as a whole, conveys the same information as the solution's "Root
→ cause" or "Diagnosis chain" even in different words.
4. Triage each finding:
 - Externally observable without tools -> tier 1, keep in symptom.md, reword if the phrasing leaks.
 - Obtainable with runtime tools AND does not reveal the fix location -> tier 2, extract into
→ `cases/<case-name>/evidence/`.
 - Requires source knowledge OR reveals fix location or strategy -> tier 3, redact entirely.
5. Extract tier 2 findings into evidence files (e.g., `stack-trace.md`, `logs.md`, `metrics.md`, `config-state.md`,
→ `runtime-behavior.md`, `test-output.md`, `error-output.md`), each with a "How this was obtained" section
→ (standard runtime tool, not source reading), an "Evidence" section (raw tool output), and a "What this tells a
→ developer" section (one to two sentences of direction, without naming the fix or the buggy function). Sanitize
→ stack traces by preserving full shape (call depth, thread or goroutine count) while redacting frames whose name
→ alone would identify the buggy function.
6. Rewrite `symptom.md` to contain tier 1 only. Replace internal data structure names with observable consequences,
→ protocol phase names with user visible events, and regression references with version ranges. Append a single line
→ pointer to the new evidence directory if evidence files were created.
7. Re run both the lexical and the semantic scans against the rewritten `symptom.md` and every evidence file. The
→ symptom file must have zero violations. Evidence files must have zero tier 3 violations. Apply the litmus tests:
→ grep test, diagnosis test, fresh developer test, narrowing test, regression diff test; plus, for evidence, the fix from
→ evidence, curation, and interpretation tests.
8. Report: a per case table of findings (status, type, tier, action taken), the list of evidence files created, a summary
→ count per tier, and the remaining violation count (must be zero).

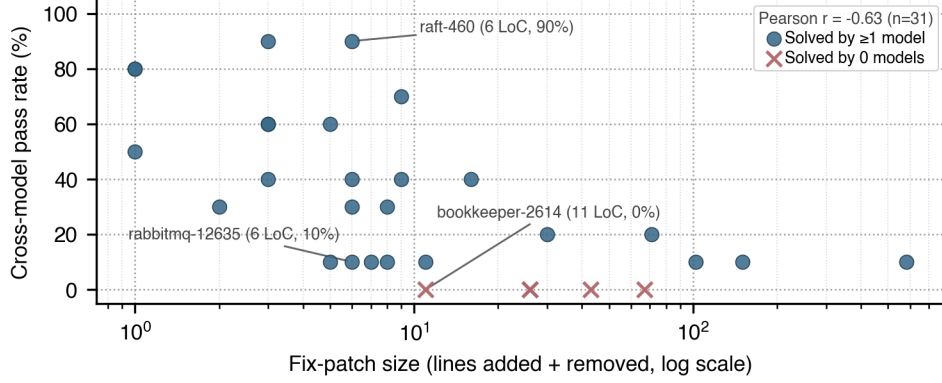


FIGURE 8: Fix-patch size vs. cross-model pass rate on tier-1 ($n = 31$). Each marker is a case; LoC is the upstream production-file diff size (lines added + removed, tests/mocks excluded), sourced from `cases-solution/*/solution.md` (packaged with DDBENCH) or from the upstream GitHub PR diff when not annotated directly in DDBENCH, with the smallest passing agent patch as a fallback (the executable oracle accepts only correct fixes, so passing-patch size faithfully bounds upstream scope). Red $\times$ markers denote cases unsolved by every model. Patch size correlates only moderately with difficulty ($r = -0.63$); within the 6-LoC bucket alone, cross-model pass rate ranges from 10% to 90%, confirming that reasoning load, not the surface size of the fix, drives difficulty.

D Benchmark Features

D.1 Difficulty: Reasoning, Not Patch Size

The difficulty of DDBENCH does not come from the size of the fix. We analyzed the size of the upstream fix patches for DDBENCH’s tier-1 case-set and found most upstream patches are surgical: median 8 lines added plus removed across tier-1. A manual review of these patches reveals that the underlying issues are typically a misplaced ordering, a missing fence, or a violated invariant rather than large-scale, multi-file code errors. The difficulty comes from the path required to *reach* the fix: the agent must reconstruct cross-process causality from incomplete logs, reason about non-deterministic interleavings, and validate hypotheses against protocol-level invariants that source-code reasoning alone cannot easily capture. Figure 8 quantifies this observation by plotting cross-model pass rate against fix-patch size. An 11-line BookKeeper recovery-fencing fix is solved by zero of ten models, while a 6-line Raft log-cache ordering fix is solved by nine; within that same 6-LoC bucket, a RabbitMQ quorum-queue snapshot crash is solved by only one, showing an 80 pp difference on patches of identical size. This is the reasoning mode that drives our headline result, producing a wide cross-model spread despite surgical patches. Single-process repair benchmarks structurally cannot exercise this reasoning mode on LLMs.

D.2 Bounded Debug-Context Channel

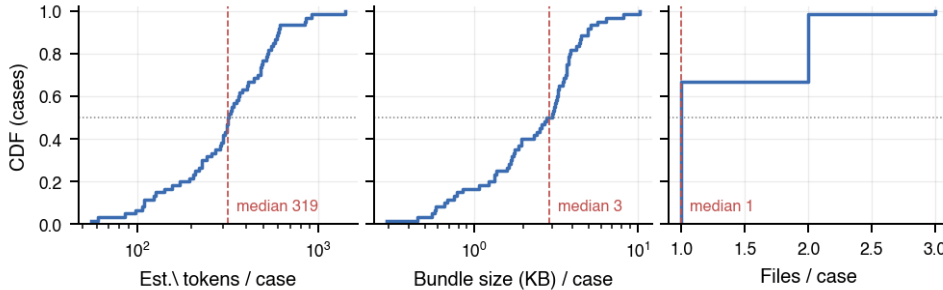


FIGURE 9: Per-case debug-context bundle size across all 60 cases. Red dashed lines mark the median; the horizontal grey line is the 50th percentile. The channel is bounded across every case: median 321 estimated tokens, max 1,403 tokens, ≤ 3 markdown files per case.

Figure 9 plots the per-case empirical CDF of debug-context-bundle size across all 60 cases along three axes: estimated tokens (using a $1.3\times$ word-count proxy that approximates common tokenizers), bundle size in kilobytes, and file count. The distribution is heavy-tailed but tightly bounded. Specifically, the median is 321 tokens with a maximum of 1,403, median 3.0 KB with a maximum of 10.4 KB, and at most three markdown files per case. Even the largest bundle occupies a small fraction of every evaluated model’s context window, so model differences in the context-augmented condition cannot be attributed to differential context-budget pressure.

Table 6 reports the file-type composition: code investigation appears in 55.0% of cases, test output in 41.7%, and crash output in 20.0%, with the remainder split among structured runtime artifacts (goroutine dumps, race-detector and sanitizer reports, GDB backtraces, and runtime state captures). Most cases combine one or two of these categories, mirroring what a developer would realistically capture during a short triage session.

D.3 DDBENCH

Position Relative to Existing Benchmarks.

TABLE 6: Debug-context-bundle file-type composition (60 cases). Categories are not mutually exclusive, so shares do not sum to 100%.

Category	n	Share
Code investigation	33	55.0%
Test output	25	41.7%
Crash output	12	20.0%
Runtime state	3	5.0%
Race/sanitizer report	3	5.0%
Debugger trace (GDB)	2	3.3%
Goroutine dump	1	1.7%

TABLE 7: DDBENCH versus existing code-repair and IT-operations benchmarks. Existing code-repair benchmarks supply an executable oracle but restrict to single-process workloads; IT-operations benchmarks exercise cross-service reasoning but ask agents to diagnose from telemetry rather than repair source. DDBENCH is the only benchmark that combines an executable repair oracle, cross-process bugs, and a controllable runtime-evidence channel.

Benchmark	Cross-process	Code fix	Executable oracle	Debug-context channel	Languages
SWE-bench Verified [13]	✗	✓	✓(unit tests)	✗	1 (Python)
Multi-SWE-Bench [37]	✗	✓	✓(unit tests)	✗	8
DebugBench [22]	✗	✓	✓(unit tests)	✗	3
AIOpsLab [8]	✓	✗	✓(telemetry)	✓(telemetry)	—
CloudOpsBench [25]	✓	✗	✓(telemetry)	✓(telemetry)	—
DDBENCH (ours)	✓	✓	✓(E2E & unit tests)	✓(controlled)	5

Table 7 situates DDBENCH against the closest existing benchmarks along four axes: whether the workload requires cross-process reasoning, whether the agent is asked to produce a code fix (as opposed to only a diagnosis), whether the harness exposes a controllable debug-context channel, and the language coverage. Code-repair benchmarks, such as SWE-bench, Multi-SWE-Bench, and DebugBench, provide executable oracles but restrict to single-process workloads. IT-operations benchmarks, such as AIOpsLab and CloudOpsBench, exercise cross-service reasoning but ask agents to identify operational problems from telemetry rather than repair source code. DDBENCH is the only benchmark that combines an executable repair oracle, cross-process bugs, and a controlled debug-context channel, making the effect of debugging context on agent performance directly measurable.

E Benchmark Infrastructure

This section walks through the end-to-end flow of running a single DDBENCH case against an agent under evaluation. The flow is driven by a single orchestrator script, `bench.sh`, that exposes six verbs (`setup`, `reproduce`, `shell`, `agent`, `reset`, `clean`) and resolves each verb against a declarative `case.yaml` plus the layered Docker images described in §C.2. From the caller’s perspective, evaluating one case is a three-step sequence: bring up a hermetic container with the faulty worktree built inside it (`setup`), launch the agent against that container under either of the two information conditions (`agent`), and score the resulting patch by running the hidden oracle (`reproduce`). We describe each step in turn, then the reset and teardown path used between conditions and models.

Step 1: Container bring-up. `bench.sh setup <case>` reads `case.yaml` for the target case, resolves the project name, the repository URL, the buggy commit, the submodule flag, and the build command, and then performs four actions. First, it builds the project’s Docker image by chaining `ubuntu:24.04`, the language base image (`bug-bench/base-cpp`, `bug-bench/base-go`, and analogous images for Java, Erlang, and Rust), and the thin per-project image that adds only build dependencies. Second, it creates or reuses a per-project Docker volume (`bug-bench-<project>-src`)

that holds a bare clone of the upstream repository, so that all cases from the same project share a single clone rather than paying the clone cost per case. Third, it materializes a detached git worktree at the buggy commit under `/workspace/worktrees/<case>` inside the volume, so that cases from the same project remain isolated at their respective commits. Fourth, it runs the build command inside a throwaway container whose working directory is the worktree, writes the sanitized `SYMPTOM.md` into the worktree root, and generates a `REBUILD.sh` script tailored to the detected build system (`ninja`, `make`, `go build`, `cargo`, `mvn`, or `gradle`) so that the agent has a single, stable incremental-rebuild entry point regardless of the underlying toolchain. After setup, the worktree holds the faulty source, the built binaries, `SYMPTOM.md`, and `REBUILD.sh`.

Step 2: Agent launch under a chosen information condition. `bench.sh agent <case> -addon <agent-name> [-evidence]` starts a long-running container named after the case, installs the requested agent (Claude Code, Aider, Codex, or mini-swe-agent) via a runtime *addon* script that caches the agent’s binaries and language runtimes in a shared Docker volume (`bug-bench-tools` mounted at `/opt/tools`), and then execs the agent inside the container with its working directory pinned to the case worktree. Addons decouple agent installation from the project Docker images, so adding a new agent requires a single shell script under `addons/` rather than a rebuild of each project image. The two evaluation conditions differ in exactly one respect: whether the per-case `evidence/` folder is mounted into the worktree as a read-only directory. Under the symptom-only condition, the agent sees only `SYMPTOM.md`, the full source tree at the buggy commit, the built binaries, and `REBUILD.sh`. Under the context-augmented condition, `bench.sh agent` additionally mounts `cases/<case>/evidence/` at `<worktree>/evidence/` so that the agent can list and read the debug-context files generated by the procedure in §C.5. API keys (`ANTHROPIC_API_KEY`, `OPENAI_API_KEY`, and similar) are forwarded from the host environment, and agent configuration directories (e.g., `~/ .claude/`) are bind-mounted read-only. The agent is invoked with the instance template documented in Appendix §F. In summary, it reads `SYMPTOM.md`, inspects `evidence/`, browses and edits source files, iteratively runs `REBUILD.sh` to keep the worktree compiling, and signals task completion by printing a sentinel string. A hard wall-clock budget bounds the trajectory, and trajectory logs are written to a mounted output directory on the host for later analysis.

Step 3: Oracle-based scoring. Once the agent terminates, `bench.sh reproduce <case>` is run in a fresh container against the same worktree volume. `reproduce` first re-runs the incremental build so that any source edits made by the agent are compiled, then executes `/bench/reproduce.sh` inside the container with the case scripts bind-mounted at `/bench/` (read-only) and the built worktree at `/workspace/worktrees/<case>`. The `reproduce` script implements the per-case correctness oracle described in Appendix §C.2. It starts the system (for server-based cases), triggers the bug with a scripted client workload or an injected regression test, checks the outcome, and prints exactly one terminal line of the form `PASS: ...` or `FAIL: ...` with a matching exit code (0 for PASS, 1 for FAIL). The oracle is hermetic in two respects. First, the agent never sees `reproduce.sh` or anything under `/bench/` during its trajectory (the mount is added only at scoring time), so the agent cannot game the oracle by reading it. Second, the oracle was validated in both directions by the procedure in Appendix §C.4, so a correct patch on the buggy commit should give a PASS. Textual match against the upstream fix is *not* required. DDBENCH’s headline metric is the fraction of cases on which `reproduce` returns PASS after the agent’s trajectory. All other metrics (tokens consumed, number of actions, wall-clock latency) are derived from the trajectory logs captured during Step 2.

Step 4: Reset and cleanup between runs. Evaluating multiple models or both information conditions on the same case requires returning the worktree to its pristine buggy state between runs, without repaying the clone or build cost. For this purpose, the command `bench.sh reset <case>` is provided to revert the worktree. It preserves `SYMPTOM.md` and `REBUILD.sh`, discards every source modification with `git checkout -.` and `git clean -fd`, moves HEAD back to the recorded buggy commit if the agent moved it, restores the two preserved files, and re-initializes submodules (if applicable). The agent evaluation container and the debug-context mount are stopped and discarded, but the per-project source volume and the addon tool cache are kept so that the next agent invocation pays only an incremental-rebuild cost. `bench.sh clean <case>` removes a single case’s worktree (leaving the shared clone and other cases untouched), and `bench.sh clean-all` removes every DDBENCH volume, image, and addon cache for a full reset. Across the full benchmark, this design means that a same-project family of cases (for instance, the 14 Dragonfly cases) shares a single clone volume and a single project image, while every case still evaluates against its own isolated worktree pinned to its own buggy commit.

F DDBENCH Prompt

F.1 System Prompt

System Prompt

You are a software debugging agent working on a distributed systems codebase.

You interact with the environment by calling the bash tool.
Think carefully before each command. Be systematic and precise.

Include a THOUGHT section before your command to explain your reasoning.
Failure to follow this format will cause your response to be rejected.

F.2 Symptom-Only Condition Instance Template

Instance Template

```
<symptom_file>
A file called SYMPTOM.md in the current directory describes the bug you need to fix.
</symptom_file>

<instructions>
## Task

You are debugging a distributed systems codebase. The bug is described in SYMPTOM.md.
Your goal is to identify and fix the root cause by modifying source files in place.

{{task}}

## Constraints

- Read SYMPTOM.md first to understand the observable failure.
- You may freely browse the codebase, build, and run tests.
- You may access the internet for API docs, language references, and general knowledge.
- You MUST NOT search for the specific GitHub issue, pull request, or known fix.
  Do not search GitHub, Google, or any other source for the specific issue number or fix.
- Make your fix by editing source files directly.
- You MUST ensure ./REBUILD.sh completes successfully after your changes.
  A failing REBUILD.sh means your fix is incomplete.
- Your fix should address the root cause, not just mask symptoms.
- You should never early declare your finish unless you have very strong evidence that the bug is fixed.

## Workflow

1. Read SYMPTOM.md to understand the failure.
2. Explore the relevant source code to understand the component architecture.
3. Form a hypothesis about the root cause.
4. Implement a targeted fix (prefer minimal, correct changes).
5. If the fix doesn't solve the bug, go back to step 2. If you are confident that you have fixed the bug, go to step 6.
6. Before submitting the final output, you MUST run `./REBUILD.sh` and confirm it exits successfully.
  Fix any compile errors before proceeding. (this is not related to your bug, it is just a sanity check)
7. When you are absolutely confident that the bug is fixed, submit the job completion by running: `echo
↪ COMPLETE_TASK_AND_SUBMIT_FINAL_OUTPUT`
  Do not combine it with any other command.
  <important>After this command, you cannot continue working on this task.</important>

## Command Execution Rules

Each response should include:

1. A THOUGHT section explaining your reasoning and plan
2. Other things.

CRITICAL REQUIREMENTS:
```

- Your response SHOULD include a THOUGHT section explaining your reasoning
- Directory or environment variable changes are not persistent across commands
- You can prefix any action with `cd /path/to/dir && ...` or write/load env vars from files

Editing Files

Create a new file:

```
```bash
cat <<'EOF' > newfile.go
package main
import "fmt"
func main() { fmt.Println("hello") }
EOF
```
```

Edit files with sed:

```
```bash
Replace all occurrences
sed -i 's/old_string/new_string/g' filename.go

Replace only in specific line range
sed -i '10,20s/old_string/new_string/g' filename.go
```
```

View file content:

```
```bash
nl -ba filename.go | sed -n '10,20p'
```
```

</instructions>

F.3 Evidence-Augmented Condition Instance Template

Instance Template

<symptom_file>

A file called SYMPTOM.md in the current directory describes the bug you need to fix.

</symptom_file>

<evidence>

The `evidence/` folder in the current directory contains debugging evidence that a developer has gathered during their debugging process. This evidence may provide additional insights into the root cause of the bug.

****You MUST review the evidence early in your investigation:****

1. Run `ls evidence/` to see what files are available.
2. Read each file with `cat evidence/<filename>`.

Evidence files may contain test outputs, log snippets, stack traces, crash reports, or code-level observations that can help you narrow down the root cause more quickly.

However, treat this evidence critically: it reflects one developer's investigation and may be incomplete.

</evidence>

<instructions>

Task

You are debugging a distributed systems codebase. The bug is described in SYMPTOM.md. Your goal is to identify and fix the root cause by modifying source files in place.

{{task}}

Constraints

- Read SYMPTOM.md first to understand the observable failure.

- Review the evidence/ folder to see what debugging evidence is available.
- You may freely browse the codebase, build, and run tests.
- You may access the internet for API docs, language references, and general knowledge.
- You MUST NOT search for the specific GitHub issue, pull request, or known fix.
Do not search GitHub, Google, or any other source for the specific issue number or fix.
- Make your fix by editing source files directly.
- You MUST ensure ./REBUILD.sh completes successfully after your changes.
A failing REBUILD.sh means your fix is incomplete.
- Your fix should address the root cause, not just mask symptoms.
- You should never early declare your finish unless you have very strong evidence that the bug is fixed.

Workflow

1. Read SYMPTOM.md to understand the failure.
2. Run `ls evidence/` and `cat` each file to review the debugging clues the developer gathered.
3. Explore the relevant source code to understand the component architecture.
4. Form a hypothesis about the root cause, informed by the symptom and any evidence.
5. Implement a targeted fix (prefer minimal, correct changes).
6. If the fix doesn't solve the bug, go back to step 3. If you are confident that you have fixed the bug, go to step 7.
7. Before submitting the final output, you MUST run `./REBUILD.sh` and confirm it exits successfully.
Fix any compile errors before proceeding. (this is not related to your bug, it is just a sanity check)
8. When you are absolutely confident that the bug is fixed, submit the job completion by running: `echo`
↪ COMPLETE_TASK_AND_SUBMIT_FINAL_OUTPUT`
Do not combine it with any other command.
<important>After this command, you cannot continue working on this task.</important>

Command Execution Rules

Each response should include:

1. A **THOUGHT** section explaining your reasoning and plan
2. Other things.

CRITICAL REQUIREMENTS:

- Your response SHOULD include a THOUGHT section explaining your reasoning
- Directory or environment variable changes are not persistent across commands
- You can prefix any action with `cd /path/to/dir && ...` or write/load env vars from files

Editing Files

Create a new file:

```
```bash
cat <<'EOF' > newfile.go
package main
import "fmt"
func main() { fmt.Println("hello") }
EOF
```
```

Edit files with sed:

```
```bash
Replace all occurrences
sed -i 's/old_string/new_string/g' filename.go

Replace only in specific line range
sed -i '10,20s/old_string/new_string/g' filename.go
```
```

View file content:

```
```bash
nl -ba filename.go | sed -n '10,20p'
```
```

</instructions>

G Benchmark Results

G.1 Pairwise Model Differentiation on Distributed System Debugging

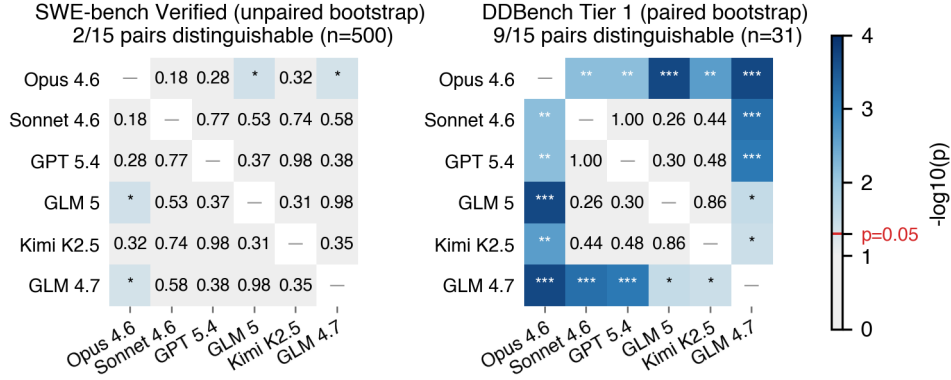


FIGURE 10: Pairwise bootstrap tests on the six models with published SWE-bench Verified scores (10,000 resamples each). DDBENCH Tier-1 uses a paired bootstrap over 31 per-case outcomes; SWE-bench Verified uses an unpaired bootstrap over 500 simulated Bernoulli trials per model at vendor-reported rates (as per-case data not publicly released). Cells show the two-sided p per pair; color encodes $-\log_{10}(p)$ clipped to $[0, 4]$, with a red mark at $p = 0.05$ ($* < 0.05$, $** < 0.01$, $*** < 0.001$). On the same six models, DDBENCH Tier-1 separates 9 of 15 pairs at $p < 0.05$, compared with 2 on SWE-bench Verified. The contrast is consistent with DDBENCH exercising a reasoning dimension that single-process code repair does not.

We present the full pairwise bootstrap matrix in Figure 10, comparing the six models with published SWE-bench Verified scores across both benchmarks (evaluated on DDBENCH tier-1 case set). SWE-bench Verified distinguishes only 2 of 15 pairs at $p < 0.05$, while DDBENCH distinguishes 9 of 15 pairs. For instance, SWE-bench Verified does not clearly separate Opus 4.6 from Sonnet 4.6 ($p = 0.18$), while DDBENCH does with $p < 0.01$.

TABLE 8: Per-model pass rates (%) on tier-2 (15 cases) under symptom-only and evidence-augmented conditions. Δ denotes the percentage-point improvement. Eight of ten models improved; two tied (Claude Sonnet 4.6, GPT OSS 120B); none regressed.

| Model | Thinking | Form | Total Cost (\$) | | | Pass Rate % | | |
|---------------------|---------------|-----------------|-----------------|-----------|---------------|-------------|-----------|----------|
| | | | Base | +Evidence | Δ (pp) | Base | +Evidence | Δ |
| Claude Opus 4.6 | Adaptive High | API | 28.46 | 22.12 | -22.3% | 73.3 | 80.0 | +6.7 |
| Claude Sonnet 4.6 | Adaptive High | API | 27.47 | 21.42 | -22.0% | 73.3 | 73.3 | — |
| GPT 5.4 | High | API | 10.75 | 10.35 | -3.7% | 60.0 | 86.7 | +26.7 |
| GPT 5.4 mini | High | API | 2.70 | 2.25 | -16.7% | 66.7 | 73.3 | +6.7 |
| GLM 5.1 (FP8) | Enabled | Open, Self-Host | 7.12 | 5.33 | -25.1% | 40.0 | 46.7 | +6.7 |
| GLM 5 | Enabled | Open, API | 36.68 | 22.01 | -40.0% | 53.3 | 66.7 | +13.3 |
| GLM 4.7 | Enabled | Open, API | 19.88 | 11.30 | -43.2% | 33.3 | 66.7 | +33.3 |
| Gemma 4 (31B) | Enabled | Open, Self-Host | 0.94 | 0.77 | -18.1% | 26.7 | 53.3 | +26.7 |
| Kimi K2.5 | N/A | Open, API | 8.30 | 2.39 | -71.2% | 33.3 | 60.0 | +26.7 |
| GPT OSS (120B BF16) | Enabled | Open, Self-Host | 1.95 | 1.65 | -15.4% | 26.7 | 26.7 | — |

G.2 Pass Rate on Tier-2 and Tier-3

We report the per-model pass rates and total cost on tier-2 (15 cases, Table 8) and tier-3 (14 cases, Table 9) under both the symptom-only and context-augmented conditions, complementing the aggregate tier comparison in §3.5.

Results on tier-2. On tier-2, 8 of 10 models improved under the context-augmented condition, 2 tied, and none regressed. The two ties are Claude Sonnet 4.6 at 73.3% and GPT OSS (120B BF16) at 26.7% in both conditions. GLM 4.7 posted the largest gain at +33.3 pp, doubling its base pass rate from 33.3% to 66.7%. Three further models gained +26.7 pp: GPT 5.4 rose from 60.0% to 86.7%, Gemma 4 (31B) from 26.7% to 53.3%, and Kimi K2.5 from 33.3% to 60.0%. GPT 5.4’s 86.7% is the highest pass rate on tier-2 under either condition, consistent with the pattern on tier-1 where frontier capability combined with debug context produces the strongest joint lift. Total cost fell for every

TABLE 9: Per-model pass rates (%) on tier-3 (14 cases) under symptom-only and evidence-augmented conditions. Δ denotes the percentage-point improvement. Six of ten models improved; two tied (GLM 4.7, GLM 5); two regressed.

| Model | Thinking | Form | Total Cost (\$) | | | Pass Rate % | | |
|---------------------|---------------|-----------------|-----------------|-----------|---------------|-------------|-----------|----------|
| | | | Base | +Evidence | Δ (pp) | Base | +Evidence | Δ |
| Claude Opus 4.6 | Adaptive High | API | 30.56 | 14.61 | -52.2% | 85.71 | 92.86 | +7.14 |
| Claude Sonnet 4.6 | Adaptive High | API | 21.32 | 10.47 | -50.9% | 85.71 | 92.86 | +7.14 |
| GPT 5.4 | High | API | 13.17 | 9.50 | -27.9% | 78.57 | 92.86 | +14.29 |
| GPT 5.4 mini | High | API | 2.37 | 1.75 | -26.2% | 85.71 | 92.86 | +7.14 |
| GLM 5.1 (FP8) | Enabled | Open, Self-Host | 9.47 | 10.41 | +9.9% | 57.14 | 50.00 | -7.14 |
| GLM 5 | Enabled | Open, API | 32.78 | 23.70 | -27.7% | 85.71 | 85.71 | — |
| GLM 4.7 | Enabled | Open, API | 20.67 | 15.35 | -25.7% | 85.71 | 85.71 | — |
| Gemma 4 (31B) | Enabled | Open, Self-Host | 1.55 | 2.75 | +77.4% | 57.14 | 50.00 | -7.14 |
| Kimi K2.5 | N/A | Open, API | 12.01 | 3.87 | -67.8% | 42.86 | 85.71 | +42.86 |
| GPT OSS (120B BF16) | Enabled | Open, Self-Host | 1.98 | 1.83 | -7.6% | 35.71 | 42.86 | +7.14 |

model on tier-2, ranging from -3.7% on GPT 5.4 to -71.2% on Kimi K2.5. GLM 5, GLM 4.7, and Kimi K2.5 each realized cost reductions above 40%, the strongest compression of the exploration phase across the cohort.

Results on tier-3. On tier-3, the picture shifts toward saturation. 6 of 10 models improved, 2 tied at 85.71%, and 2 regressed by -7.14 pp each. The two ties are GLM 5 and GLM 4.7. The regression GLM 5.1 corresponds to a single case flip at $n = 14$ within binomial noise. Under the context-augmented condition, four models converge at 92.86% pass rate, Claude Opus 4.6, Claude Sonnet 4.6, GPT 5.4, and GPT 5.4 mini, reflecting the ceiling effect expected on this tier since tier-3 cases are predominantly single-process, input-triggered crashes whose root cause often localizes within one file. The largest gain comes from the weaker end of the cohort. Kimi K2.5 climbed $+42.86$ pp from 42.86% to 85.71%, reaching the open-weight cluster anchored by the GLM 5 and GLM 4.7 tie point. Cost trajectories are more mixed than on tier-2. 8 of 10 models reduced total cost, with Kimi K2.5 dropping -67.8% and the two Claude models dropping approximately 50%, while Gemma 4 (31B) roughly doubled its cost as the extra passes it now converts come from cases it previously abandoned early. GLM 5.1 and GPT OSS (120B BF16) saw small cost shifts of $+9.9\%$ and -7.6% respectively, tracking the same pattern where debug context enables longer but more productive trajectories on previously unsolved cases.

Takeaway. The per-model tables reinforce the aggregate trend in §3.5. Debug context lifts pass rate broadly but with a narrower margin as tier difficulty decreases, and the cost benefit is not uniform. Models already near the ceiling realize most of the benefit as cost reduction, while weaker models realize it as pass-rate gains, often accepting higher cost in exchange.

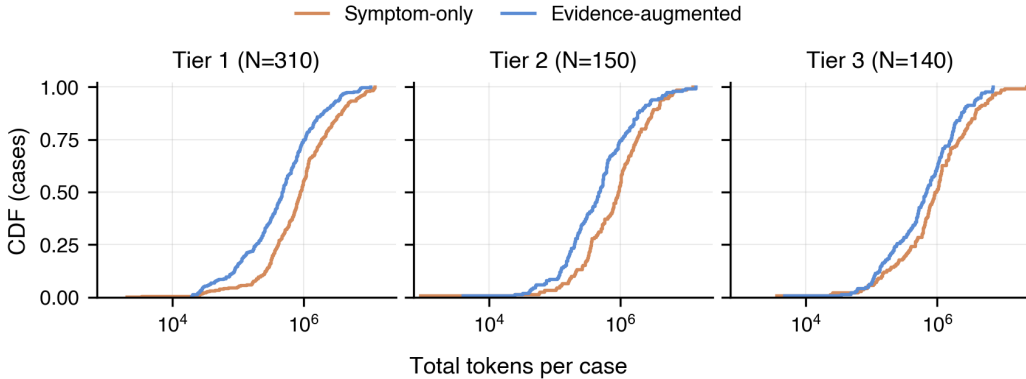


FIGURE 11: Per-case total-token CDFs on tier-1–tier-3. Each panel overlays the symptom-only (orange) and context-augmented (blue) distributions, pooling all evaluated models. Medians shift from 0.88M to 0.47M on tier-1 (-47%), 0.93M to 0.47M on tier-2 (-49%), and 0.95M to 0.68M on tier-3 (-28%). The context-augmented CDF lies to the left of the symptom-only CDF at every percentile on tier-1 and tier-2; on tier-3 the two curves converge in the upper tail.

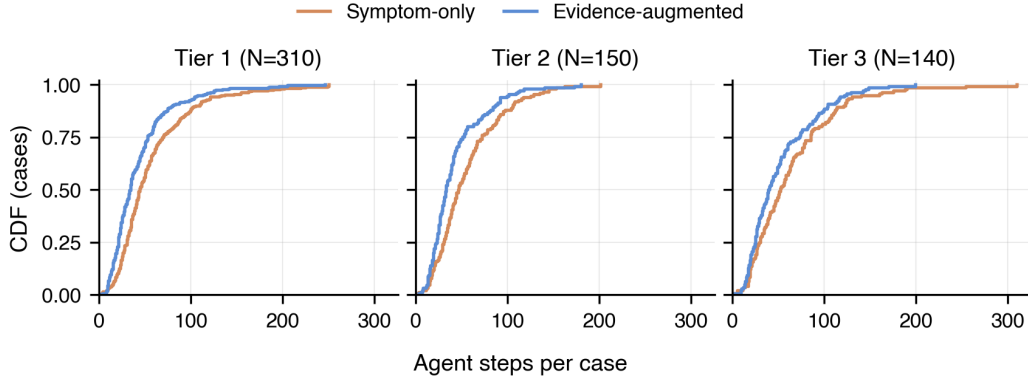


FIGURE 12: Per-case agent-step CDFs on tier-1–tier-3. Medians drop from 44 to 34 on tier-1 (−23%), 46 to 33 on tier-2 (−28%), and 52 to 40 on tier-3 (−23%). Unlike the token distributions, the step CDFs separate roughly uniformly across the support, so the savings are not driven by a few long-running outliers.

G.3 Efficiency Gain from Debug Context

The trajectory analysis (§3.4) reported mean steps and tokens on tier-1. Figure 11 and Figure 12 extend the result along two axes. We plot the full per-case distribution rather than the mean, and we repeat the measurement on tier-2 and tier-3 to validate that the effect is not tier-specific.

Tokens. Debug context cut the median per-case token budget by nearly half on tier-1 (0.88M $\rightarrow$ 0.47M) and tier-2 (0.93M $\rightarrow$ 0.47M), and by 28% on tier-3 (0.95M $\rightarrow$ 0.68M). The context-augmented CDF dominates the symptom-only CDF on tier-1 and tier-2: at every quantile a context-augmented run consumed fewer tokens than the matched symptom-only run. On tier-3, the two curves cross in the upper tail. Specifically, a small number of cases ran to the step budget in both conditions, indicating that when the abstraction distance between the supplied symptom and the fault is large (§3.3), debug context does not shorten the exploration loop.

Agent steps. The step distributions shifted more uniformly than tokens: medians dropped 23–28% across tiers, and the two CDFs sit roughly parallel. The per-tier median reductions (tier-1: −10 steps; tier-2: −13; tier-3: −12) are consistent with the previously reported mean of −13.9 steps on tier-1 (Table 3). Debug context therefore removes a near-constant number of exploratory steps per case, rather than only trimming the long tail.

Why the CDF matters. The mean-based summary in Table 3 leaves open whether the efficiency gain is driven by a handful of pathological runs or by a broad shift. Figure 11 and Figure 12 resolve this by showing that the shift is distributional and persists across tiers. Combined with the paired-outcome analysis (§G.5), which shows efficiency gains on concordant-PASS, concordant-FAIL, and FAIL $\rightarrow$ PASS cases, these CDFs support the conclusion that debug context acts as a search-space reducer for agent diagnosis, narrows the set of hypotheses the agent must explore, and is independent of whether the final patch changes.

G.4 Per-Model Step-Count Distributions

Figure 12 pools per-case step counts across the full cohort, which leaves open whether the median reduction reflects a uniform per-model shift or is driven by a subset of the cohort. Figure 13–Figure 17 decompose this view by plotting the per-case step histogram for each of five models that span the cohort’s capability range, pooling tier-1–tier-3 cases per model so that each panel carries enough mass to estimate the distribution shape. We restrict each panel to runs that terminated with the Submitted exit status so that the histogram reflects deliberation length rather than abnormal termination (step-cap or context-window exceptions, or wall-clock timeouts). The two side-by-side panels in each figure share a step axis and overlay the per-condition median as a dashed reference inside each panel.

Cross-model synthesis. Three patterns emerge from the per-model panels. First, the median reduction is universal in this subset: every model shows a leftward shift under context augmentation, ranging from −4 steps for GPT 5.4 to −25 steps for Claude Opus 4.6. This rules out the hypothesis that the cohort-level effect in Figure 12 is carried by a few frontier models; lighter models show the

Opus 4.6 (tier 1-3, completed cases)

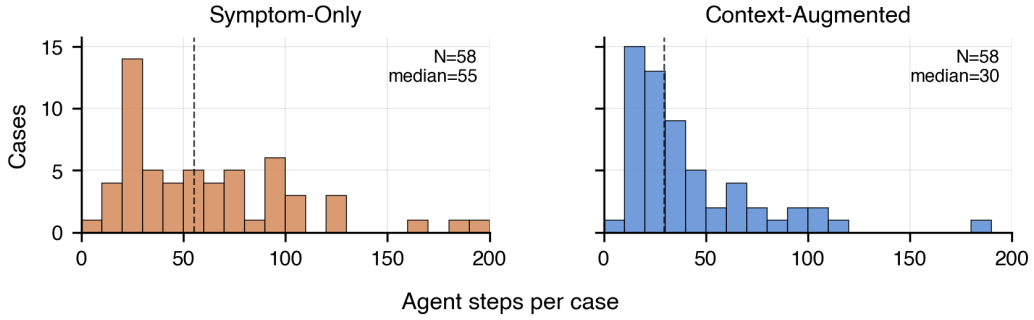


FIGURE 13: Per-case agent-step distribution for Claude Opus 4.6 across tier-1–tier-3, restricted to runs that terminated under Submitted. The median drops from 55 to 30 steps under context augmentation ($\approx 45\%$ reduction). The right tail extends past 180 steps in both conditions—the longest in the cohort—indicating that even the strongest model in our setup encounters cases where the cross-process search dominates the trajectory regardless of debug context.

Sonnet 4.6 (tier 1-3, completed cases)

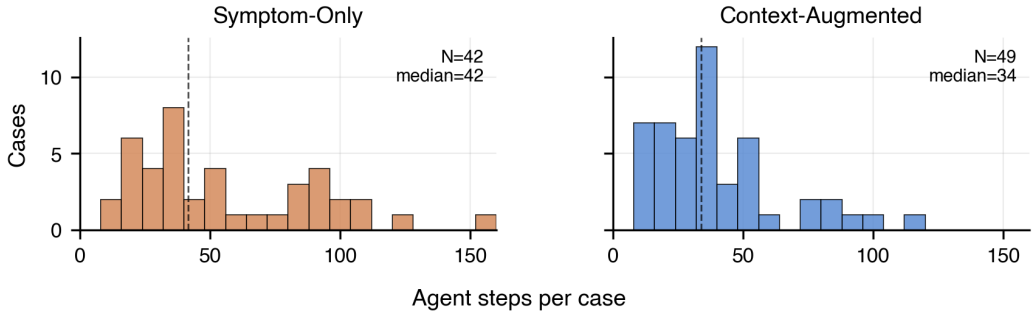


FIGURE 14: Per-case agent-step distribution for Claude Sonnet 4.6 across tier-1–tier-3. The median drops from 42 to 34 steps. The base count ($N = 42$) is lower than the context-augmented count ($N = 49$): seven additional cases reached Submitted under context augmentation. The recovered cases are dominated by runs that produced no exit status under the symptom-only condition (most plausibly wall-clock timeouts or framework aborts) rather than by step-cap exceptions; under either reading, the asymmetry is consistent with debug context shortening the path to a candidate patch and is not visible in the median alone.

GPT 5.4 (tier 1-3, completed cases)

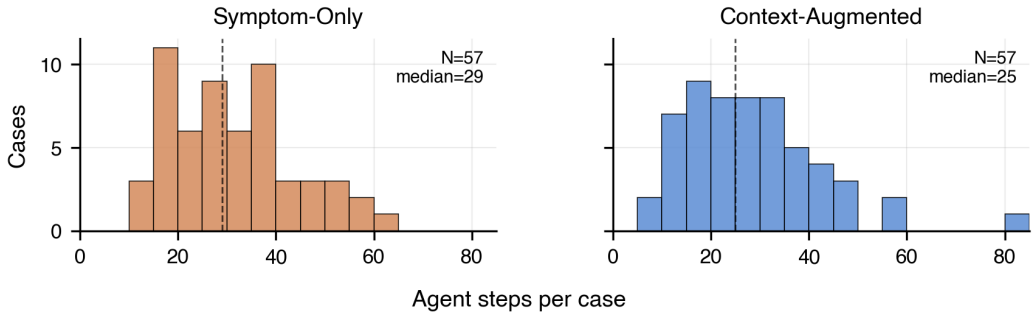


FIGURE 15: Per-case agent-step distribution for GPT 5.4 across tier-1–tier-3. The median drops from 29 to 25 steps. Both distributions concentrate tightly below 85 steps with negligible right-tail mass, indicating that GPT 5.4 either converges on a candidate patch quickly or abandons the case, rather than continuing to explore.

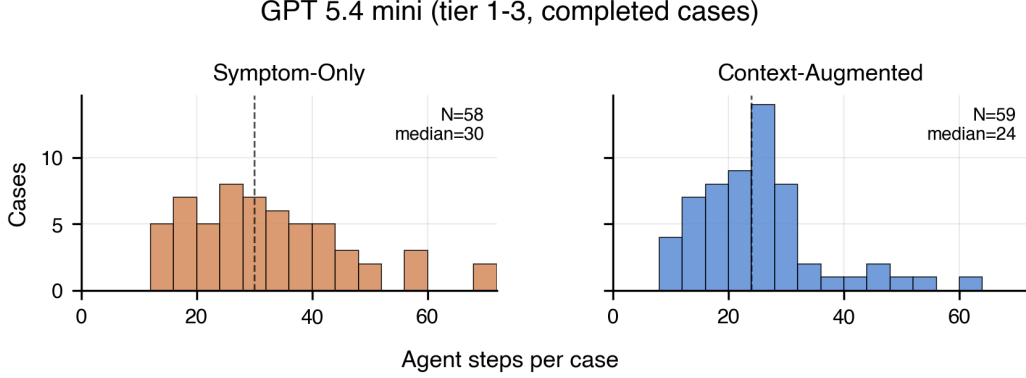


FIGURE 16: Per-case agent-step distribution for GPT 5.4 mini across tier-1–tier-3. The median drops from 30 to 24 steps. The shape mirrors GPT 5.4: short-tailed and tightly concentrated. The smaller absolute reduction (−6 steps) is consistent with the already-compact base distribution—there are fewer wasted exploratory steps for debug context to remove.

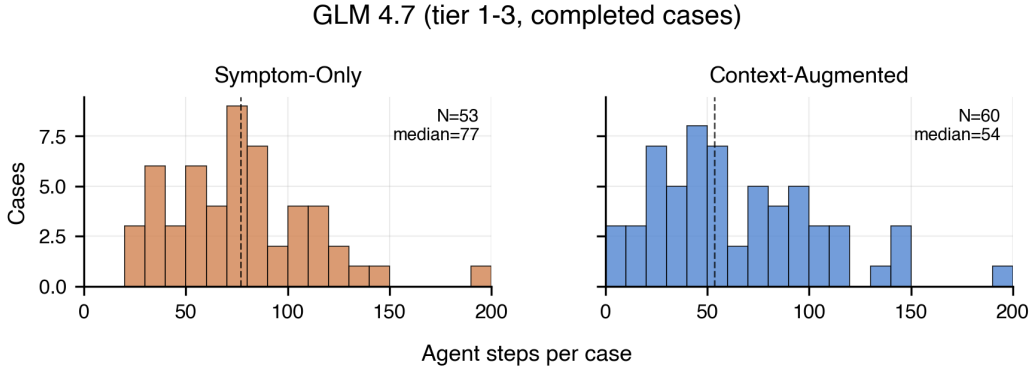


FIGURE 17: Per-case agent-step distribution for GLM 4.7 across tier-1–tier-3. The median drops from 77 to 54 steps—the highest medians in this subset under either condition. The right tail extends past 190 steps in both conditions, reflecting GLM 4.7’s tendency to run a long exploratory phase on hard cases. The base count ($N = 53$) is lower than the context-augmented count ($N = 60$): all seven cases that did not terminate under Submitted in the symptom-only condition (one step-cap exception and six runs with no recorded exit status) reached Submitted when given debug context, so every GLM 4.7 case terminated normally under context augmentation.

same direction even when the magnitude is small. Second, the absolute reduction tracks the base-condition median. Models that already settle quickly under the symptom-only condition (the GPT 5.4 family) have less slack to compress, while models that explore extensively under symptom-only conditions (Claude Opus 4.6, GLM 4.7) realize the largest absolute step savings. This is consistent with the search-space-reduction reading from §G.3: debug context removes exploratory steps, and a model that takes few exploratory steps in the first place gains less. Third, the heavy right tails are concentrated in Claude Opus 4.6 and GLM 4.7, both of which run long deliberation loops on hard cases rather than abandoning early; the GPT 5.4 family in contrast caps trajectories well below 85 steps in both conditions, suggesting an earlier give-up policy. These two failure modes—long-tail persistence vs. early abandonment—are visible at the distribution level even when the medians are similar, and they suggest that scaffold-design work aiming to extract more from a fixed step budget should treat the two regimes separately.

G.5 Per-case Paired Outcome in Tier-1 under Debug-Context Augmentation

We present the full per-case paired outcomes on tier-1 under debug-context augmentation in Figure 18. Weak models (bottom rows) are dominated by green gains, where GLM 4.7 flips 13 of 31 cases. Strong models (top rows) are dominated by blue tie-passes. Losses are rare and concentrated on

a small set of cases where the runtime symptom misdirects. For example, dragonfly-6687 (the second column of Resource & delivery) is the highlighted example analyzed in §3.3.

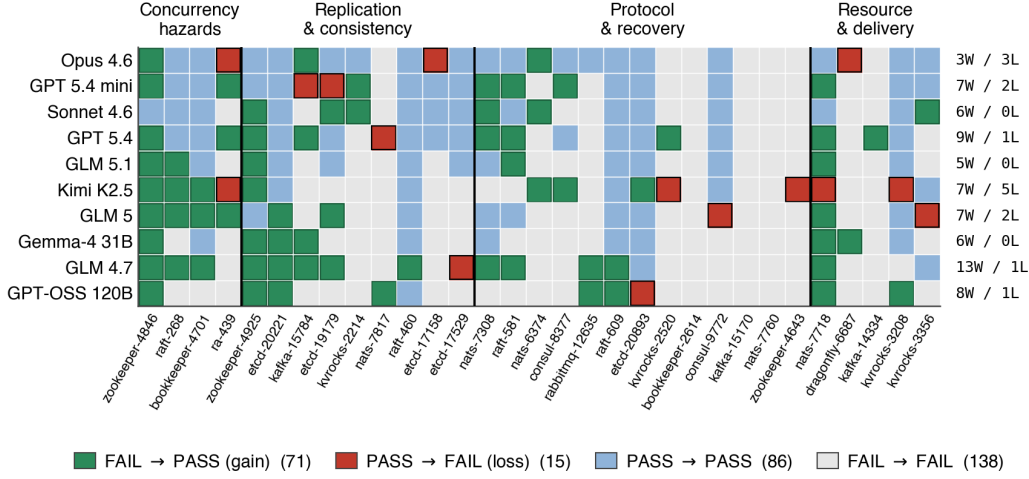


FIGURE 18: Per-case paired outcomes across all 10 models on tier-1 (rows: models, ordered by symptom-only pass rate; columns: the 31 cases, grouped by bug category and ordered within each group by cross-model win count). Green cells are gains (FAIL→PASS), red cells are regressions (PASS→FAIL). Weak models (bottom rows) are dominated by green—GLM 4.7 flips 13 of 31 cases—while strong models (top rows) are dominated by blue tie-passes. Losses are often cases where the runtime symptom misdirects. For instance, dragonfly-6687 (second column of Resource & delivery) is one of the highlighted examples in Appendix §G.6.

G.6 Case Studies: When Debug Context Helps, and When It Misleads

This section expands the qualitative analysis summarized in §3.3. We examine three tier-1 cases that together span the qualitative regimes of the debug-context effect: one in which debug context supplies a signal unreachable from code alone (ZooKeeper-4846), one in which debug context compresses a tractable but expensive search (Raft-268), and one in which debug context misleads the agent across an abstraction gap (Dragonfly-6687). The cases were selected by cross-model consensus, so that a majority of the 10 models show the same directional effect and the pattern reflects task structure rather than noise from any single model.

ZooKeeper-4846: debug context supplies a signal that no code reading can recover (9/10 FAIL→PASS). The bug corrupts access-control state after a crash-recovery path, so that child nodes are re-linked to stale permission entries whenever the on-disk snapshot and the in-memory state disagree. Under the symptom-only condition, 9 of 10 models failed, and not because they did not try: agents explored the snapshot and restore code paths broadly, yet the fault only manifests as a specific post-recovery runtime state that static reading of the codebase cannot reconstruct. When we provided the post-recovery permission table as debug context, 9 of 10 models succeeded and converged on a precise fix in `createNode()`. The behavioral shift is consistent with an information-gap hypothesis. Kimi K2.5’s exploration actions dropped from 9 to 2, its read actions from 26 to 8, and its first-edit step from 46 to 17. Gemma 4, which had given up after 4 steps without ever editing, now navigated directly to the relevant file and patched the bug in 12 steps. The near-unanimous flip across a heterogeneous model set indicates that debug context here acts as a capability unlock: when the root cause is a runtime-only invariant, the missing ingredient is signal rather than reasoning ability.

Raft-268: debug context compresses a tractable but expensive search (4/10 FAIL→PASS, 0 regressions). The bug is a shutdown-path deadlock, in which one goroutine blocks on a channel that a second, already-stopped goroutine was supposed to service. The root cause is in principle derivable from the source, but doing so requires reasoning over the full space of goroutine interleavings, a search that scales badly with context length. The debug context we provide is a goroutine stack dump that points directly at the blocked operation, which collapses this combinatorial reasoning problem into a localized one. With debug context, 4 additional models flip to PASS and no model regresses. The more revealing effect, however, is on trajectory rather than outcome. Kimi K2.5

went from 225 steps (first edit at step 174) to 36 steps (first edit at step 9); its symptom-only patch was a surface-level workaround in the wrong file, whereas its context-augmented patch correctly restructured the blocking `select`. Even Claude Opus, which passed under both conditions, halved its step count (58 to 30) and its first-edit step (51 to 21). Debug context here functions as a search-space reducer, and we argue this is where most of the aggregate trajectory effect in §3.4 originates, since the efficiency benefit is visible even when the final outcome does not change.

Dragonfly-6687: debug context misleads across an abstraction gap (1 win, 1 loss). The root cause of this bug is a race in the *serialization* path, in which two writers interleave into a shared byte stream and produce a corrupted output. The runtime symptom, however, surfaces much later in the *deserialization* path on a downstream replica, as a cryptic decoding error. The debug context faithfully reports this symptom, and in doing so anchors the agent to the wrong abstraction layer. Claude Opus was the only model to solve this case under the symptom-only condition, through 190 steps of code exploration and commit-history inspection; once given debug context, it was misdirected into the decoder and produced an incomplete fix, flipping the outcome from PASS to FAIL. Conversely, Gemma 4 moved from a no-edit failure to a partial workaround near the error location, an improvement in engagement but still short of the true fix. Here, debug context inherits the abstraction level of the observable symptom, and when the symptom sits several layers above the cause, debug context acts as a confident but wrong prior rather than a neutral input. This is not a quirk of a single case: the 8/10 cross-model failure rate under both conditions indicates that such abstraction gaps mark the current capability frontier, and symptom-proximal debug context alone is unlikely to move it.

NeurIPS Paper Checklist

1. Claims

Question: Do the main claims made in the abstract and introduction accurately reflect the paper’s contributions and scope?

Answer: [Yes]

Justification: We provide a dataset, benchmark infrastructure, and a thorough evaluation on them that collectively reflect the technical, conceptual, and major contributions of this paper. The claims and findings mentioned in abstract and introduction are thoroughly supported and discussed in the later sections (§2 and §3). The benchmark shape is fully described in §2, and the evaluation results are fully described in §3. The claims on the model differentiations on distributed debugging and controlled study of symptom-only and context-augmented are fully discussed in §3, which further reveals several findings that is backed by the evaluation results. All number that is mentioned in the abstract and introduction are accurate and also supported by the evaluation results in §3.

Guidelines:

- The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
- The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
- The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
- It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. Limitations

Question: Does the paper discuss the limitations of the work performed by the authors?

Answer: [Yes]

Justification: We fully discussed the limitations of our work in §4, including the limitations of the evaluation (single agent scaffold, no content-level ablation) and the single curated release. We also fully discussed the potential future work to address these limitations, including multi-scaffold evaluation, content-level ablation, and scale-up across additional systems and bug families.

Guidelines:

- The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
- The authors are encouraged to create a separate “Limitations” section in their paper.
- The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
- The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
- The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
- The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
- If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
- While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren’t acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that

preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

Answer: [N/A]

Justification: This work does not include theoretical results and we do not have any theorems to prove, so this question is not applicable.

Guidelines:

- The answer [N/A] means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

Answer: [Yes]

Justification: We fully disclosed all the information needed to reproduce the main experimental results in §3, including the details of the models evaluated, the agent scaffold used, and the evaluation protocol. We also provided the details of the benchmark features and data curation process in §2. Appendix §E also includes the details of data mining and prompts we used for the semi-agentic mining process. We released the codebase, dataset, and benchmark infrastructure (including the semi-agentic mining tools) to facilitate reproducibility.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
- If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
- Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general, releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
- While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
 - (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
 - (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
 - (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).

- (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. Open access to data and code

Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

Answer: [Yes]

Justification: We released the codebase, dataset, and benchmark infrastructure (including the semi-agentic mining tools) to facilitate reproducibility. The dataset and codebase are also submitted alongside the paper and will be made publicly available upon publication. We also provided detailed instructions for how to use the codebase and reproduce the main experimental results in the README and documentation.

Guidelines:

- The answer [N/A] means that paper does not include experiments requiring code.
- Please see the NeurIPS code and data submission guidelines (<https://neurips.cc/public/guides/CodeSubmissionPolicy>) for more details.
- While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
- The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (<https://neurips.cc/public/guides/CodeSubmissionPolicy>) for more details.
- The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
- The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
- At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
- Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. Experimental setting/details

Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

Answer: [Yes]

Justification: This paper does not include training details as we do not train any models, but we provide all the details of the experimental setting and evaluation protocol in §3, including the models evaluated, the agent scaffold used, and the evaluation protocol. We had dedicated subsections in evaluation section describes the model we used, the conditions (symptom-only vs. context-augmented) we evaluated, and the empirical bounds we set for evaluating agent performance (bounded max steps, time limits, and token cost).

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
- The full details can be provided either with the code, in appendix, or as supplemental material.

7. Experiment statistical significance

Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

Answer: [Yes]

Justification: Every claim that supports the headline contributions is accompanied by an explicit uncertainty estimate, computed by a method documented in §3.1. Pairwise model-differentiation

results (Figure 10) are obtained by a paired case-level bootstrap over the 31 tier-1 cases (10,000 resamples) and reported as a two-sided p -value per model pair with 95% CIs on the pass-rate difference; the SWE-bench Verified comparison panel uses the analogous unpaired bootstrap over per-model Bernoulli draws since per-case outcomes are not publicly released. Per-model debug-context effects use the same 10,000-resample paired bootstrap (§3, “Statistical significance”), and the aggregate debug-context effect is estimated by a mixed-effects logistic regression with random intercepts for model and case over all 620 paired outcomes, reported with point estimate, standard error, 95% CI, odds ratio, and Wald z -statistic. The factors of variability captured by these intervals are the random sampling of cases (bootstrap) and the model $\times$ case grid the regression is fit on. All intervals are two-sided 95% unless otherwise stated.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
- If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. Experiments compute resources

Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

Answer: [Yes]

Justification: We fully described the containerized evaluation infrastructure which implicitly specifies the compute resources required. As we did not train any models in this paper, so we do not have any strong requirement on the type of compute workers. Any commodity CPU server with sufficient memory and storage should be able to run this benchmark.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
- The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
- The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn’t make it into the paper).

9. Code of ethics

Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics <https://neurips.cc/public/EthicsGuidelines>?

Answer: [Yes]

Justification: Yes, the entire research process, methodology and results fully conform to the NeurIPS code of ethics. We have carefully reviewed the code of ethics and ensured that our research adheres to all the guidelines and principles outlined in it. We have taken necessary

precautions to ensure that our research does not cause harm, respects privacy, and is conducted with integrity and transparency. We did not conduct any experiments involving human subjects or crowdsourcing, so we did not need to obtain IRB approval or equivalent.

Guidelines:

- The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
- If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
- The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

Answer: [Yes]

Justification: We do not anticipate any direct negative societal impacts of our work, but we do discuss the potential positive impacts and insights throughout the evaluation section §3. We discussed the potential future work on enhancing agent engineering and model capabilities for distributed debugging, which could have positive impacts on software engineering practices and productivity in general.

Guidelines:

- The answer [N/A] means that there is no societal impact of the work performed.
- If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.
- Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

Answer: [N/A]

Justification: We do not anticipate any risk of misuse of our dataset, benchmark, or evaluation results, so we do not have any specific safeguards in place for responsible release. However, the case used in this benchmark are all sanitized and runs inside an isolated container by default.

Guidelines:

- The answer [N/A] means that the paper poses no such risks.
- Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
- Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.

- We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.
12. **Licenses for existing assets**
- Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?
- Answer: [Yes]
- Justification: We properly credited the creators and original owners of all assets used in the paper, including the code repository, datasets, and literature. We also included a dedicated section in Appendix §B.2 that lists all licenses of code repositories that is included in this benchmark release.
- Guidelines:
- The answer [N/A] means that the paper does not use existing assets.
 - The authors should cite the original paper that produced the code package or dataset.
 - The authors should state which version of the asset is used and, if possible, include a URL.
 - The name of the license (e.g., CC-BY 4.0) should be included for each asset.
 - For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
 - If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, paperswithcode.com/datasets has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
 - For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
 - If this information is not available online, the authors are encouraged to reach out to the asset's creators.
13. **New assets**
- Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?
- Answer: [Yes]
- Justification: The codebase for benchmark infrastructure is well documented with a README and detailed instructions for how to use the codebase and reproduce the main experimental results. The cases curated in this release are also all well documented with detailed descriptions of each case in both human-readable and machine-readable formats.
- Guidelines:
- The answer [N/A] means that the paper does not release new assets.
 - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
 - The paper should discuss whether and how consent was obtained from people whose asset is used.
 - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.
14. **Crowdsourcing and research with human subjects**
- Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?
- Answer: [N/A]
- Justification: We did not conduct any crowdsourcing experiments or research with human subjects, so this question is not applicable.
- Guidelines:
- The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
 - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.

- According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

Answer: [N/A]

Justification: We did not conduct any research with human subjects, so this question is not applicable.

Guidelines:

- The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
- Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
- We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
- For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

Answer: [Yes]

Justification: We used semi-agentic mining protocol to mine the cases in this benchmark, which involves LLMs and agents to semantically parse the information from GitHub issues and PRs and synthesize the artifacts used later in the evaluation. The full details are described in §2 and Appendix §C. Our usage of LLMs does not impact the core methodology, scientific rigor, or originality of the research, but it is an important component of the data curation process, so we declare it here for transparency.

Guidelines:

- The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
- Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.