Runtime evidence under tight tool-call budgets

Trace-augmented agent solvers on enterprise-style bugs preserve diagnostic capability where static analysis collapses

Kevin Gilpin · Elizabeth Lawler

June 2026

Contents

Abstract

Trace-augmented agent solvers can use a structured runtime trace of operating code as evidence for Root Cause Analysis (RCA); conventional static-analysis solvers must assemble the same evidence via manual exploration (`Read`/`Grep`/`Bash`). We isolate the impact of these two approaches by decomposing an agentic solver into explicit rca and fix stages, evaluating on a multi-module Java codebase and sweeping the RCA tool-call budget across {unlimited, 10, 5, 3} for two frontier models (Claude Sonnet 4.6 and Haiku 4.5) at $N = 8$. On two admitted fixtures, the trace-augmented solver (`appmap-rca`) retains capability that the static baseline (`code-rca`) loses as the budget tightens: verify-pass rates hold flat at 94–100% on `appmap-rca` but degrade monotonically from 84% (unlimited) to 62% ($b = 3$) on `code-rca`, doubling the cross-lane gap (+16 pp to +32 pp). RCA-correctness diverges more sharply — `appmap-rca` maintains 100% across all budgets while `code-rca` collapses from 91% to 28% — because the static lane exhausts its tool-call budget before assembling a diagnosis: a single runtime `get_call_tree` provides the information density a static grep trajectory needs 15+ calls to replicate. Holding accuracy at tight budgets also makes `appmap-rca` the cheapest route to a given pass rate. On the same two-fixture sweep, Haiku + `appmap-rca` at a tight budget ($b = 3$) reaches 88% verify for \$0.37/cell — the cheapest route to that accuracy — and where both lanes reach 100% verify the `appmap-rca` lane gets there for ~3.4× less spend. Extending the comparison to 11 fixtures, all-Haiku at $b = 3$ is 3.8× cheaper than the strongest static baseline (Sonnet + uncapped `code-rca`) but trails it by 21 pp (74% vs 95% verify); a hybrid pipeline — Haiku's $b = 3$ RCA report paired with a Sonnet fix step — recovers two-thirds of that gap, landing within 7 pp of the baseline (88% vs 95%) for 2× less spend. Runtime traces let compact models under strict call caps approach uncapped, frontier-model static baselines, and pairing a cheap trace-RCA model with a reliable fix model nearly closes the remaining gap.

1. Introduction

The standard benchmark for autonomous coding agents, SWE-bench Verified [1, 2], is valuable for benchmarking overall agent capability but ill-suited for studying the mechanics of root cause

analysis (RCA). Its 500 issue/patch pairs come from twelve open-source Python libraries that already sit in frontier models' pretraining corpora, confounded by model priors: a result can reflect the agent's familiarity with these specific codebases rather than its actual performed root-cause analysis. The issue descriptions also tend to state the bug mechanism and even the code location to fix, bypassing the exploratory RCA phase and leaving the agent with only an implementation task. This gap matters because the agents being measured — systems that ingest a bug report, explore a codebase, and generate a patch — have moved from research prototypes into mainstream software engineering workflows, and RCA is the step the standard benchmark exercises least.

This study isolates the root-cause analysis step to answer a fundamental, pre-empirical question: **how does the availability of runtime-trace evidence alter an agent's trajectory, accuracy, and resource consumption during root-cause analysis and subsequent patch generation?** Rather than asserting an unbacked claim of superiority, we treat this as a structural study-design question, mapping out the precise cost-capability frontier of runtime data versus traditional static code exploration.

1.1 Decoupled Experimental Lanes

To isolate the value of execution data, we explicitly decouple the agent's workflow into two sequential, independent stages: `rca` (diagnosis) and `fix` (implementation). We evaluate agent performance across two distinct solver strategies:

- **Static Baseline (`code-rca`):** The agent reads the issue description and explores the codebase using conventional static tools (`Read`, `Grep`, `Bash`) to assemble a root-cause analysis report. A separate, isolated execution session then reads this report to apply a patch.
- **Trace-Augmented Lane (`appmap-rca`):** The agent runs the same two-stage rca → fix pipeline under identical inputs, prompts, and tool-call budgets, but is additionally given an interactive, structured runtime execution trace (via Model Context Protocol tools) generated by running a reproducing integration test under AppMap instrumentation.

Neither lane's RCA agent is permitted to see the source code of the integration test itself (§2.3). The structured execution trace is the sole differentiator between the two lanes, ensuring that any variation in downstream success is directly attributable to the presence of runtime evidence.

1.2 Experimental Design Variables

Our evaluation framework controls the execution environment across five core axes to characterize agent behavior under varying constraints:

- **Lane:** Comparison between the static analysis lane (`code-rca`) and the trace-augmented lane (`appmap-rca`).
- **Model Tier:** Evaluation across distinct capability tiers using Claude Sonnet 4.6 (frontier) and Claude Haiku 4.5 (compact), both executing within the Claude Code agent environment (v2.1.159).

- **RCA Tool-Call Budget:** A cap on the number of tool invocations allowed during the RCA phase, swept across `{unlimited, 10, 5, 3}` calls to measure degradation under resource pressure.

Each cell in the resulting grid is run $N = 8$ times to estimate within-cell stochastic variance.

1.3 Summary of Contributions

By testing these configurations against tool call budget and model sweeps, this paper maps out the exact trade-offs between static exploration and dynamic tracing. In doing so, we make the following contributions:

- **The Information-Density Effect:** We demonstrate that a single runtime trace provides far higher information density per tool call, allowing agents to capture execution paths in one step that require fifteen or more sequential grep/read iterations using static analysis tools.
- **Characterization of Budget Collapse:** On the two admitted fixtures, static agents fail under strict tool constraints: `code-rca` RCA-correctness falls from 91% to 28% as budgets tighten, while appmap-rca holds at 100% (128/128 trials).
- **Definition of the Cost-Capability Frontier:** We chart the cheapest route to a target verify-pass rate. On a two-fixture sweep, the trace-augmented lane (`appmap-rca`) reaches a given rate for 3.4–3.6× less spend than the static baseline. Across 11 fixtures the edge holds with a caveat: a compact model (Haiku) at a $b = 3$ cap costs 3.8× less than an uncapped frontier model (Sonnet) but trails its verify rate by 21 pp; pairing that capped Haiku diagnosis with a Sonnet fix step recovers most of the gap, landing within 7 pp of the baseline for 2× less spend.

2. System Architecture & Experimental Lanes

To isolate the value of runtime data during program repair, we establish a controlled environment that decouples problem diagnosis from patch implementation. This section details the technical surface of our instrumentation, the capabilities provided to the agent, and the structural design of our experimental lanes.

2.1 Instrumentation & The MCP Tool Surface

We use AppMap to capture execution telemetry. The system attaches to the runtime environment via a Java agent (`-javaagent`) during JUnit integration test execution, outputting structured `.appmap.json` artifacts. Each fixture's reproducing test exercises the system end-to-end through its external interface: wherever the failure allows, it is driven entirely by HTTP requests, so it mimics real user behavior rather than calling internal methods directly. The recorded trace therefore spans the full request-to-persistence path, which is what lets a single call-tree query surface the bug mechanism (§6.2). These recordings capture fine-grained execution metadata, including function entry/exit states, argument values, return types, elapsed time, call tree hierarchy, executed SQL statements (with bound parameters), HTTP request/response lifecycles, and thrown exceptions.

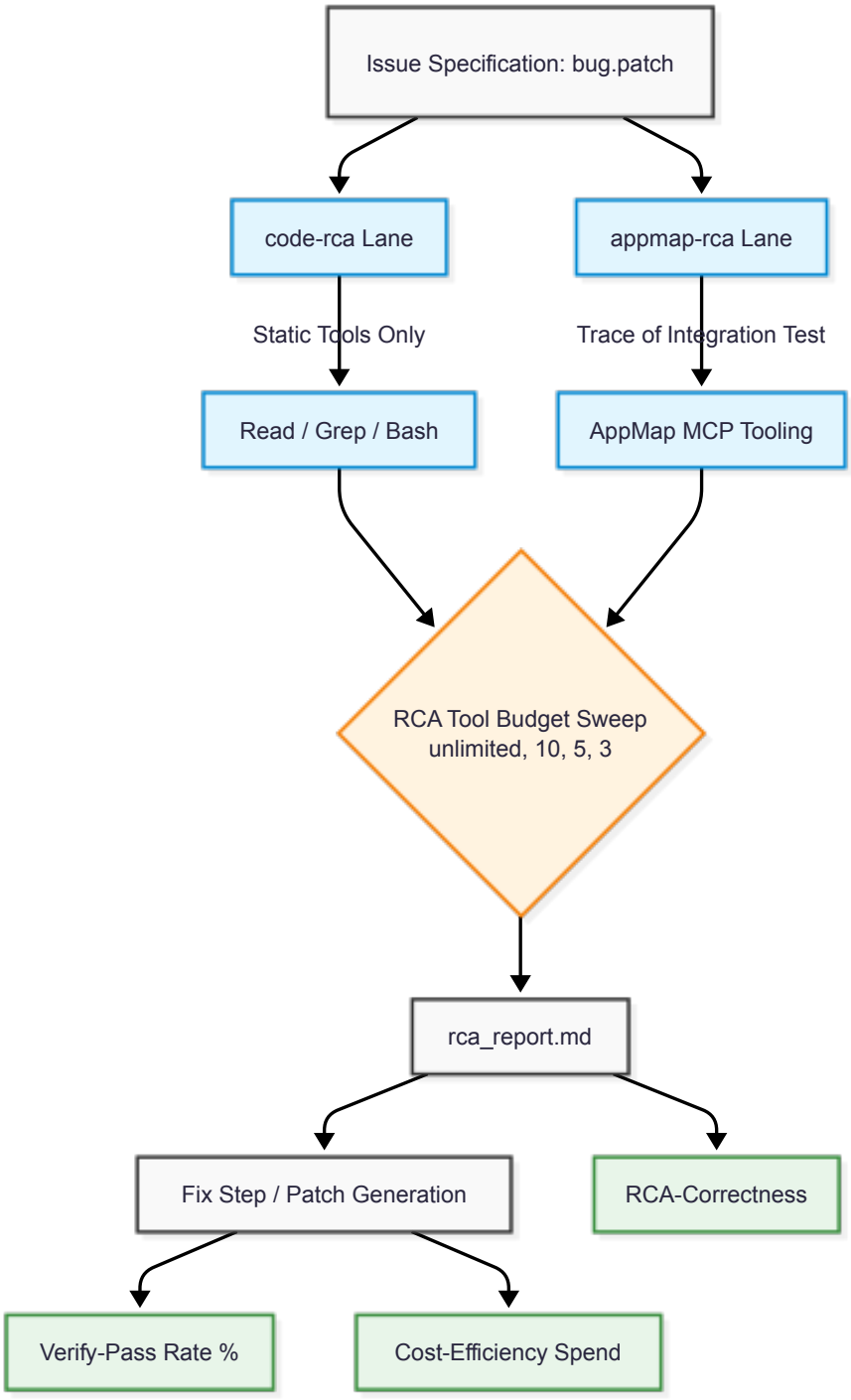


Figure 1: RCA Tooling for Bug Patch

The agent interacts with these recordings through Model Context Protocol (MCP) tools. Rather than forcing the model to parse raw JSON payloads, the MCP surface exposes discrete, high-density queries:

- `find_recordings`: Lists available trace archives, filterable by test name.
- `get_call_tree`: Returns a structured, textual call tree for a specific recording. Default fetch depth and character limits are applied; these can also be tuned by the LLM, and filter & focus parameters can be specified.
- `find_calls`: Locates all invocations of a fully-qualified function name across the active recording set.
- `find_queries`: Isolates executed SQL queries alongside their binding context.
- `find_requests`: Identifies HTTP network boundaries within the execution trace.

2.2 Experimental Solver Lanes

Our evaluation compares two distinct operational lanes. Both configurations receive an identical, symptom-only bug report (`issue.md`) and use identical underlying prompts, varying exclusively by the tool surface made available to the agent during the diagnostic phase.

1. The Static Baseline (`code-rca`)

1. **RCA Step:** The agent reads `issue.md` and explores the codebase using traditional, read-only static analysis utilities (`Read`, `Grep`, `Glob`, `Bash`). It concludes by generating a standalone text file: `rca_report.md`.
2. **Fix Step:** A separate, isolated agent session ingests the `rca_report.md`, conducts independent source exploration if necessary, and generates a functional codebase patch (`diff.patch`).

2. The Trace-Augmented Lane (`appmap-rca`)

1. **Instrumentation Step (`apply_it`):** The harness applies a reproducing integration test and executes it under the AppMap Java agent to generate runtime recordings.
2. **RCA Step:** The agent reads `issue.md` and receives explicit prompt access to the AppMap MCP tool surface, with the recording paths pre-resolved in its context window. It concludes by generating an `rca_report.md`.
3. **Fix Step:** Identical to the baseline lane, the fix agent ingests the report and applies a patch.

To ensure strict experimental isolation, the source code of the integration test itself is completely withheld from the `appmap-rca` agent; it is deleted from both the working tree and the upstream paths before the LLM session initializes. Consequently, the agent cannot read the test file, forcing it to rely entirely on the runtime trace metadata.

3. Methodological Framing: Beyond SWE-bench

The current standard for evaluating autonomous software engineering agents is SWE-bench Verified [1, 2]. However, we argue that SWE-bench possesses structural characteristics that make it an ineffective instrument for isolating the contribution of root-cause analysis evidence channels. We address these limitations through a specialized evaluation design.

3.1 Overcoming Contamination and Solution Leakage

Our departure from the SWE-bench paradigm is driven by two confounding variables inherent to its dataset design:

1. **Pretraining Data Contamination:** SWE-bench issues are drawn from twelve highly visible, open-source Python repositories (e.g., `django`, `scikit-learn`). These codebases predate the training cutoff dates of modern frontier models by several years and exist at high densities within their pretraining corpora. When an agent evaluates a familiar repository, it relies heavily on memorized structural priors (naming conventions, historical fix patterns, layout topologies), which mutes the experimental contrast between static and dynamic evidence gathering.
2. **Solution Leakage in Issue Texts:** Because SWE-bench issues are harvested from real GitHub threads post-triage, they are written by and for developers. As noted by Aleithan et al. in their characterization of *SWE-Bench+* [3], roughly one-third (33.0% of Verified) of these instances exhibit "solution leakage," where the exact file path, stack trace, or buggy function name is explicitly provided in the issue description. Recent audits, such as *The SWE-Bench Illusion* [4], confirm that models can recover the correct buggy file path from issue text alone up to 76% of the time on the Verified cohort.

For an experiment designed to compare root-cause analysis strategies, solution leakage is fatal. If an issue description discloses the location or mechanism of the bug, the RCA phase becomes degenerate; both the static baseline and the trace-augmented lane will converge on the same diagnosis by construction.

3.2 The Omnibank Testbed & Inclusion Criteria

To re-establish a meaningful measurement of raw engineering exploration, we evaluate our agents against **Omnibank**, a multi-module enterprise Java application (~50 modules, Spring Boot) authored specifically for this study. Because Omnibank is hosted in a private repository, it is completely absent from any frontier model's pretraining corpus, eliminating repository-level memorization priors.

We compose the evaluation suite with an admission pipeline that applies two criteria. The first is inherited from how Omnibank issues are written; the second is the measurement gate specific to this study's primary budget sweep.

- **No solution leakage.** Each bug report is written as a non-technical customer or operations

ticket (e.g., *"Customers being charged twice on retried payments"*), with zero identifier overlap between the issue narrative and the `gold_fix.patch` (the token-disjointness rule C1, whose construction is detailed in the methods paper). This removes the leakage that makes the RCA phase degenerate (§3.1).

- **Performance corridor.** A fixture is admitted to the primary sweep only if it sits between two rails. It must *fail* under a blind, zero-RCA-budget configuration (the agent reasoning from priors alone, with no evidence-gathering calls), so a bug solvable without any investigation is excluded as trivial. It must *pass* under an unlimited budget, so a bug no agent can solve even unconstrained is excluded as impossible. A fixture solvable at unlimited budget necessarily reproduces at runtime, which also gives the `appmap-rca` lane a trace to consume.

Fixtures that pass at every budget or fail at every budget pin the measurement rails and hide the budget-driven knee the study measures, so both are excluded. Separately, fixtures that fail the token-disjointness filter (where a structural clue is unavoidable in the issue text) are set aside into a *selectivity-control cohort* for future work (§7).

3.3 Scope and Trade-offs

We acknowledge two core trade-offs resulting from this experimental design choice. First, we forfeit direct cross-paper comparability; our headline performance numbers cannot be mapped against public SWE-bench leaderboards. Second, our focus on pristine evaluation conditions limits our current scale. While SWE-bench Verified spans 500 instances, this paper evaluates a dense, 256-cell execution matrix ($N = 8$ replications across our budget and model sweep). We accept these limitations in exchange for an unconfounded, high-fidelity measurement of agent behavior under resource pressure.

4. Methods

This section details our experimental matrix, the rationale behind our fixture selection, and the methodology used to compare trace-augmented agents against static baselines under strict resource constraints. The entire study is orchestrated via an automated execution harness, enforcing budget thresholds via runtime hooks.

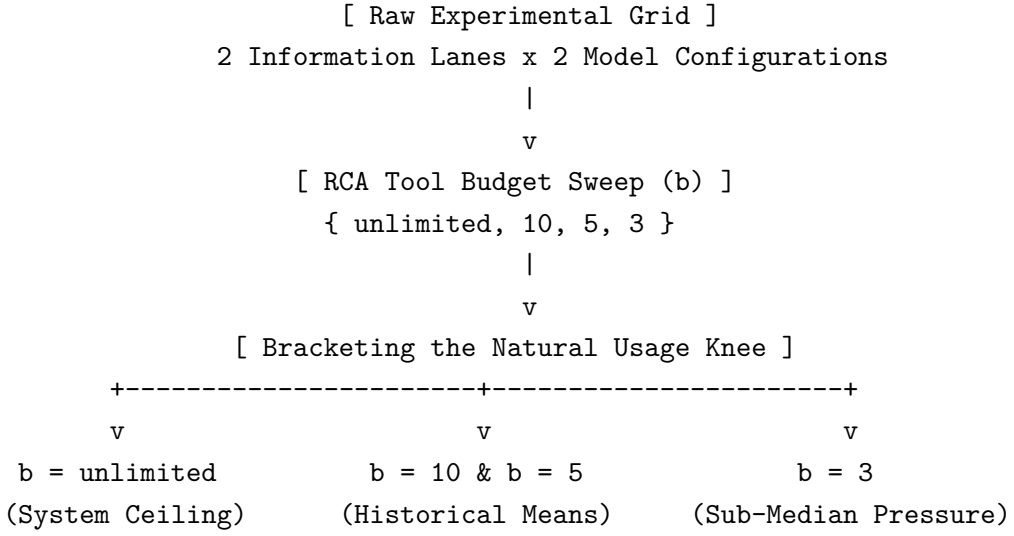
4.1 The Primary Budget Sweep Matrix

The headline study isolates the impact of diagnostic resource constraints by varying a single experimental parameter: the Root Cause Analysis (RCA) tool-call budget. We sweep this budget across four discrete levels:

$$\text{RCA Tool Call Budget } (b) \in \{\text{unlimited}, 10, 5, 3\}$$

The fix-step tool budget is held at `unlimited` across all configurations, ensuring that any measured degradation in downstream patch success is entirely attributable to resource pressure during the

diagnostic phase. While codebase read operations are strictly capped during the RCA phase, file write operations remain permanently unbounded during the patch phase so as not to prevent the agent from producing its final fix.



These four budget levels are explicitly chosen to bracket the per-model "natural-usage knee" measured during our May 2026 baseline evaluations:

- **`unlimited`**: Our experimental control ceiling: no RCA cap, so each lane runs to its own natural stopping point rather than a fixed call count. The May 2026 baselines put the aggregate natural-usage knee at roughly 10–11 calls, which motivated the `10` cap below. This study's unconstrained cells show that knee is strongly lane-dependent, however: appmap-rca self-limits at a mean of ~5 RCA tool calls (max 6 across all cells), while code-rca runs to a mean of 16–19 (max 30) (§5.1). The level is an untruncated baseline for both lanes.
- **`10` and `5`**: Bracket the historical baseline cap and the mean token trail for successful, unconstrained Claude Sonnet trials.
- **`3`**: Establishes severe resource pressure, sitting strictly below the natural median tool-call volume for both evaluated model tiers.

The budget constraints are hard-enforced at the execution runtime via a custom `PreToolUse` hook embedded within the Claude Code (v2.1.159) agent loop. When the monotonic tool counter hits the assigned boundary b, the hook intercepts the request and terminates the diagnostic session, forcing the agent to output its `rca_report.md` using only the evidence compiled up to that exact execution frame. The primary budget sweep is executed as a dense grid across both information lanes, two model tiers (Claude Sonnet 4.6 and Claude Haiku 4.5), and two contrasting fixtures. Evaluated at a sample density of $N = 8$ replications to capture stochastic variance, this matrix comprises 256 discrete trajectories.

4.2 Outcome Metrics

Each trial yields two outcomes, one objective and one graded.

Verify-pass is the end-to-end result. The fix step's patch (`diff.patch`) is applied to the fixture, and the fixture's hidden bench test is run; the trial passes when that test passes. Neither agent ever sees this test, so verify-pass measures whether the pipeline produced a working fix rather than whether it matched a known answer. A cell's verify-pass rate is the fraction of its $N = 8$ trials that pass.

RCA-correctness scores the diagnosis on its own, independent of whether a patch shipped. An LLM grader (`analysis/rca_llm_grade.py`, defined in the methods paper §5.2) compares the agent's `rca_report.md` against the gold fix's mechanism description and returns `correct`, `partial`, or `wrong`. We report the `correct` rate: the fraction of a cell's trials judged `correct`. One grader and one threshold score both lanes; we do not switch metrics between them.

The `correct`/`partial` boundary is soft. The methods paper (§5.2) notes that small differences in report phrasing can tip a verdict there, so per-fixture `correct`-rate measurements at $N = 8$ carry grader-internal variance even when report quality is steady. We report the `correct` rate regardless, for two reasons. Applying a single grader and threshold to both lanes makes that noise symmetric: it moves both lanes together, so the cross-lane gap (the quantity the study turns on) is not biased by where the grader draws the line. And on the trace lane the boundary never comes into play: `appmap-rca` is judged `correct` on 128 of 128 trials across the two fixtures, with nothing sitting near the `partial` line. The variance touches only the static lane's mid-range cells, where it widens the error bars without changing the direction of the gap.

4.3 Diagnostic Fixture Selection

Rather than selecting fixtures for broad repository coverage, the primary budget sweep targets two specific enterprise environments drawn from the curated Omnibank suite (§3.2): `omnibank_exposure-as-of-mismatch` and `omnibank_status-stuck-received`. These fixtures were selected because they independently satisfy a critical experimental precondition and test the budget constraints from polar opposite directions:

1. Verification Corridor Sufficiency

Both fixtures satisfy the §3.2 performance corridor: 0% verify-pass (0/8) under the blind, zero-budget configuration, rising to 100% (8/8) under an unlimited budget. This wide corridor is what makes a budget-driven knee visible to measure.

2. Isolation of Priors-Resistance

The selected pair sits on opposite ends of the architectural failure spectrum, isolating distinct engineering vulnerabilities:

- **Fix-Step-Resistant (`exposure-as-of-mismatch`):** LLM structural priors allow the agent to correctly identify the root cause roughly 88% of the time, but the resulting patch must span five separate files across two decoupled modules. In the `appmap-rca` lane, the runtime call tree exposes each builder's independent `asOfWatermark` capture natively. This fixture

tests how effectively the high-density trace mechanism relieves downstream implementation bottlenecks under tight diagnostic caps.

- **RCA-Step-Resistant (`status-stuck-received`):** Model priors systematically misdiagnose this bug, applying a write-path assumption to an underlying read-path defect. Breaking the agent out of this decoy behavior requires direct structural evidence. This fixture measures the exact volume of tool-call evidence an agent needs to override its erroneous structural priors.

Fixture Target	Focus Layer	Baseline Prior Tendency	Core Experimental Stress Test
`exposure-as-of-mismatch`	Fix-Step-Resistant	High Accuracy (~88%)	Measures trace-driven relief of multi-module patch bottlenecks.
`status-stuck-received`	RCA-Step-Resistant	Systemic Misdiagnosis	Measures tool-call volume required to override decoy priors.

4.4 Suite-Wide Generalization Paradigm

To test whether the behaviors captured in our two-fixture sweep generalize across wider enterprise topologies, we implement a secondary, suite-wide experiment. This run-set addresses a practical question for agent architecture design: **does a trace-augmented compact model constrained to a tight budget match or exceed a frontier static model running with an unlimited tool budget?**

We pit the trace lane at its absolute resource floor (`appmap-rca` at $b = 3$) against the static baseline at its most generous (`code-rca` at `unlimited`): the standard configuration deployed by code-only practitioners.

Deliberate Omission of the Corridor Filter

Unlike the primary budget sweep, this suite-wide evaluation is *not* filtered based on the 0/8 to 8/8 verification corridor. Instead, we deliberately retain rail-pinned fixtures across the full methodology-admitted suite (§3.4) because they cleanly isolate two distinct analytical claims:

- **The Cost Regime (High-Rail Fixtures):** On fixtures where the static baseline at unlimited budget already achieves a 100% verification rate (e.g., `org-transfer-stale-cache`), accuracy is held constant by construction. These fixtures yield an unconfounded economic measurement: accuracy is fixed, and the moving variable is financial spend. We directly map the cost optimization of `appmap-rca` at $b = 3$ against the unconstrained static baseline cost.
- **The Capability Regime (Low-Rail Fixtures):** On fixtures where the static baseline fails entirely even with an unlimited budget (e.g., `balance-walker-order-dependent`), the experiment tests pure capability recovery. A successful patch by a 3-call trace-augmented agent

over an unconstrained static failure constitutes a definitive capability result that overrides simple cost deltas.

Evaluation Protocol

To minimize unnecessary compute expenditure, we reuse the unconstrained baseline dataset from our historical reference run-set (`2026-05-28-methods-upper-bound-n8`) for the `code-rca @ unlimited` baseline. We execute the `appmap-rca @ b=3` lane anew at an identical density of $N = 8$ across both Claude Sonnet 4.6 and Claude Haiku 4.5.

As with all other arms of this study, the source code of the integration test is completely withheld from the RCA agent (§2.3), isolating the structured execution trace as the lone evidence channel under evaluation. Because the aggregate results across a diverse enterprise suite are expected to be heterogeneous, findings are reported broken down per fixture across separate cost and capability vectors rather than collapsed into an obscured global average.

5. Results

The primary axis of evaluation measures the cross-lane performance of verification pass rates (`verify-pass`) and root-cause analysis correctness (`RCA-correctness`) under systematically constrained diagnostic environments. Our evaluation dataset covers a 256-cell primary budget sweep alongside an expanded 11-fixture generalization suite.

We structure our empirical findings around two core research questions:

- **RQ1 (Accuracy under Constraint):** How does tightening the diagnostic tool-call budget affect the accuracy and correctness of trace-augmented agents versus static baselines?
- **RQ2 (The Cost-Capability Frontier):** Can trace-augmented, resource-constrained agents optimize total financial expenditure while maintaining competitive patch success rates across complex systems?

5.1 RQ1: Accuracy and Diagnostic Correctness under Budget Constraints

To evaluate the resilience of each evidence channel under resource pressure, we analyze agent behavior across our primary two-fixture sweep (`exposure-as-of-mismatch` and `status-stuck-received`), two Claude models (sonnet 4.6, haiku 4.5), and four budget levels (unlimited, 10, 5, 3) at N=8 per cell (256 cells in total), the *cross-lane budget sweep* (§8).

Our baseline measurements reveal that the trace-augmented lane (`appmap-rca`) preserves a flat verification pass rate of 94–100% across the entire budget axis. Conversely, the static baseline (`code-rca`) degrades monotonically, dropping from an unconstrained ceiling of 84% to 62% under

a tight resource constraint ($b = 3$). Consequently, the cross-lane capability gap doubles under pressure, widening from +16 percentage points (pp) at an unlimited budget to +32 pp at $b = 3$.

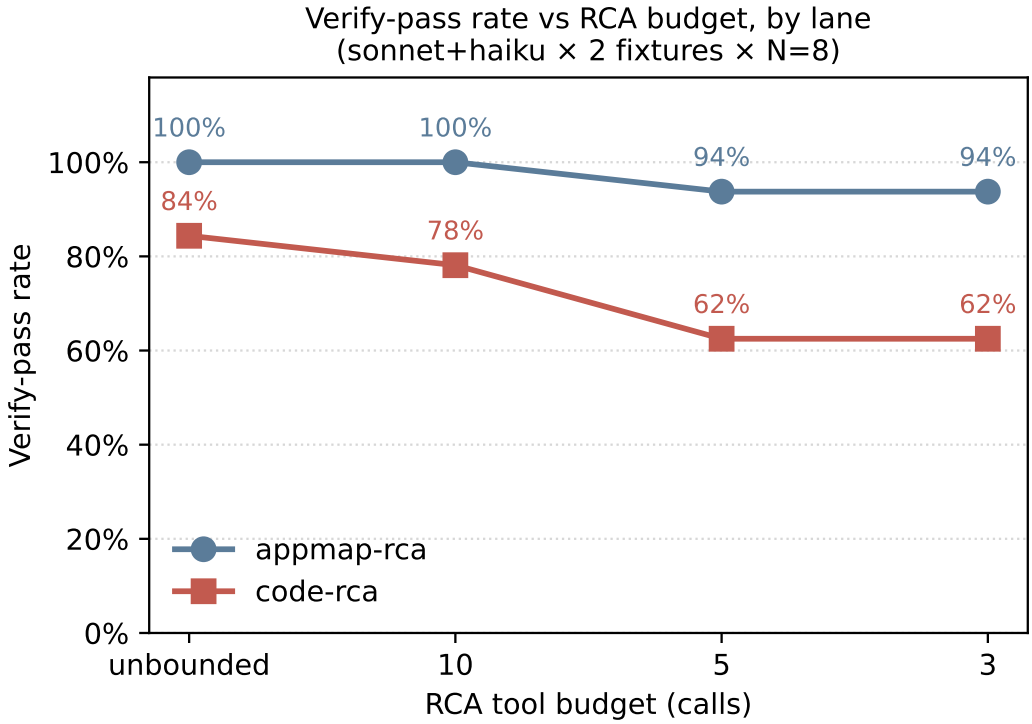


Figure 2: Verify-pass rate vs RCA budget, by lane (sonnet+haiku × 2 fixtures × N=8). The leftmost tick, "unbounded," realizes very different call volumes per lane: appmap-rca self-limits at a mean of ~5 RCA tool calls (max 6 across all cells), while code-rca runs to a mean of 16–19 (max 30).

The unconstrained execution baseline highlights a divergence in operational efficiency. When given an unlimited tool budget, the `appmap-rca` lane naturally self-terminates at an average of ~5 tool calls (reaching a maximum of 6 calls across all evaluated cells). In contrast, the static `code-rca` lane executes an average of 16–19 sequential tool calls (maxing at 30 calls) yet finishes with a lower baseline accuracy. Because the natural execution volume of the trace-augmented agent sits natively near our lowest cap, tightening the budget constraint imposes negligible behavior truncation on the `appmap-rca` lane.

This downstream verification gap is driven by a collapse in the accuracy of the underlying diagnosis, as illustrated by our LLM-graded correctness evaluation:

- **`appmap-rca` Correctness:** Maintains 100% across every budget level (`unlimited`, 10, 5, 3).
- **`code-rca` Correctness:** Collapses from 91% to 81%, 50%, and ultimately 28% at $b = 3$.

Under strict call caps ($b = 3$), more than two-thirds of static agent trials run out of exploration budget before successfully identifying the bug mechanism. Under identical constraints, the trace-augmented agent achieves 100% diagnostic correctness, demonstrating that structured runtime

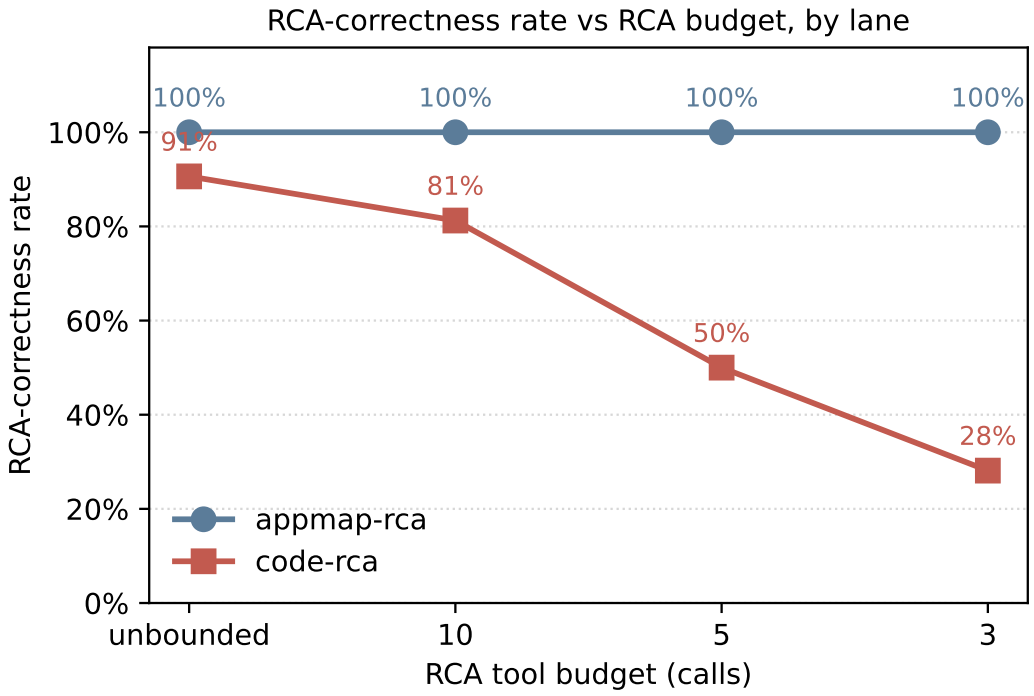


Figure 3: RCA-correctness rate vs RCA budget, by lane.

evidence isolates root causes in a single execution step where traditional text search requires dozens of iterative operations.

> **Summary of RQ1:** Runtime execution tracing insulates agent diagnostic capability against severe tool-call constraints. While static exploration models suffer an accuracy collapse from 91% down to 28% under budget constraints ($b = 3$), trace-augmented models maintain a 100% diagnostic success rate by optimizing information density per tool call.

5.2 RQ2: Economic Optimization and Suite-Wide Cost Frontiers

Because trace-augmented agents preserve structural accuracy under strict tool caps, they open up an optimal cost-capability frontier. We analyze the financial profiles of our configurations to identify the most efficient route to given target verification pass rates.

5.2.1 The Two-Fixture Cost Frontier

Within our primary two-fixture sweep, a single architectural configuration dominates the low-to-mid performance tiers: **Haiku + `appmap-rca` restricted to a strict tool cap** ($b = 3$). This configuration delivers an 88% verification pass rate at a minimal financial footprint of $0.37 per cell.

target verify	standard (sonnet + code-rca)	lowest-cost option	cost saving	RCA budget
75%	sonnet · code · b=5, $1.32 (75%)	haiku · appmap · b=3, $0.37 (88%)	**3.6× / 72%**	5 → 3
94%	sonnet · code · b=10, $1.42 (100%)	haiku · appmap · b=10, $0.42 (100%)	**3.4× / 70%**	10 → 10
100%	sonnet · code · b=10, $1.42 (100%)	haiku · appmap · b=10, $0.42 (100%)	**3.4× / 70%**	10 → 10

Reading the table: *target verify* is the verify-pass rate to reach; *standard* is the cheapest sonnet + code-rca configuration that reaches it (the conventional baseline); *lowest-cost option* is the cheapest configuration of any (model, lane, budget) that reaches it; *cost saving* is the per-cell cost ratio between the two and its percent reduction; *RCA budget* shows the standard's tool budget alongside the lowest-cost option's.

When mapped against standard deployment patterns, this pairing is a clear cost advantage over the conventional static benchmark:

- The Static Benchmark: To clear a 75% verification pass rate, a static frontier model (Sonnet + `code-rca`) requires an intermediate budget ($b = 5$), consuming $1.32 per cell.
- The Trace Alternative: Haiku + `appmap-rca` ($b = 3$) surpasses this performance threshold while operating 3.6× cheaper (a −72% cost reduction).

At the absolute performance ceiling, the economic benefit shifts from capability recovery to pure cost reduction. While both lanes are capable of reaching 100% verification pass rate, the static baseline (Sonnet + `code-rca` at $b = 10$) requires $1.42 per cell to do so. The trace-augmented alternative (Haiku + `appmap-rca` at $b = 10$) delivers that same 100% verification floor for $0.42 per cell, yielding a 3.4 × cost reduction (-70%$).

5.2.2 Generalizing the Frontier Across 11 Fixtures

To evaluate whether this economic edge scales beyond our primary configurations, we executed a suite-wide replication across 11 diverse, methodology-admitted fixtures ($N = 8$, totaling 264 trajectories), the *11-fixture cost suite* (§8).

Table 1 logs the cross-lane performance comparison.

Table 1: Comprehensive cost and capability suite-wide summary.

Model & Architecture Configuration	Verification Pass Rate (%)	Mean Financial Cost ($/cell)	Performance Delta vs. Static Baseline
Sonnet × `code-rca` × Unbounded *(Static Baseline)*	95% (83/87)	$1.161	—
Haiku × `appmap-rca` × $b = 3$ *(All-Compact Candidate)*	74% (65/88)	$0.309	−21 pp accuracy, **3.8× cheaper**
Hybrid (Haiku × `appmap-rca` $b = 3$ RCA + Sonnet Fix)	**88% (77/88)**	$0.567 ∗ ∗\| ∗ ∗$**-7 pp$ accuracy, 2.0× cheaper**	

(A per-fixture side-by-side of the three configurations is in the artifact catalog, §8.)

Our suite-wide evaluation indicates that the pure all-Haiku configuration does not survive wider enterprise topologies unmodified. When scaled to 11 fixtures, the aggregate verification rate drops to 74% (a −21 pp shortfall against the baseline). This regression is concentrated within two specific environments: `approval-tier-evasion` and `tranche-cancel-cascade`. In both scenarios, Haiku successfully generates a correct RCA report under its $b = 3$ constraint but lacks the structural capability to synthesize a functional patch during its subsequent execution step.

The Hybrid Pipeline Recovery We address this downstream bottleneck by establishing a hybrid agent architecture that preserves Haiku's low-cost $b = 3$ diagnostic reports but passes them to a frontier model (Sonnet) for the final implementation step. This deployment configuration achieves a balanced trade-off:

1. Accuracy Recovery: The hybrid pipeline closes the performance gap, capturing an 88% verification pass rate (falling within 7 pp of the unconstrained static frontier model).
2. Economic Efficiency: It retains the cost benefit, operating 2.0× cheaper than the static baseline ($0.567 vs. $1.161 per cell).

Because the underlying diagnostic reports are identical to the all-Haiku run (75% correctness), this performance recovery is entirely driven by the fix-step's capability to buffer marginal or partial RCA inputs. The mechanism is the *fix-step-buffers-RCA-quality* effect developed in §6.1.

For instance, in the `approval-tier-evasion` fixture, the hybrid configuration raises verification pass rates from 3/8 to 8/8, despite working from an RCA context that was only 5/8 correct.

Conversely, the lone edge case that resists this recovery is `tranche-cancel-cascade`. This fixture demands a complex two-phase-commit rollback implementation that a frontier model cannot gener-

ate when restricted to a tight 3-call diagnostic window. This boundary confirms that the coupling between fix-stage execution and diagnostic tool boundaries is inherently fixture-dependent, charting the final absolute limits of our low-cost hybrid recipe.

> **Summary of RQ2:** Across an expanded 11-fixture evaluation, a pure compact architecture (Haiku) cuts engineering costs by 3.8× at the expense of a 21 pp drop in verification success. By deploying a hybrid architecture (combining a budget-capped Haiku diagnosis with a frontier Sonnet fix step), system designers can recapture most of that capability, pulling within 7 pp of an unconstrained static baseline for exactly half the financial cost (2.0× cheaper).

6. Analysis

The empirical results in Section 5 establish that runtime evidence preserves diagnostic accuracy under resource pressure, thereby optimizing the economic frontier of agentic program repair. This section analyzes the underlying systemic mechanisms driving these results and outlines the boundary conditions of the study.

6.1 Downstream Buffering: Diagnosis vs. Implementation

The separation between our two evaluated metrics, RCA-correctness and verify-pass rates, exposes a critical architectural phenomenon: **downstream stage buffering**.

When the static baseline (`code-rca`) is restricted to a tight budget ($b \leq 5$), its RCA-correctness collapses. However, its final verify-pass rate declines less severely. Trajectory analysis confirms this is due to the unconstrained `fix` step actively compensating for the partial or empty diagnostic report. The frontier model uses its unlimited patch-generation budget to re-perform the root-cause analysis from scratch.

Conversely, the trace-augmented lane (`appmap-rca`) preserves the diagnosis itself. Because the initial RCA is universally correct (100% accuracy at $b = 3$), the downstream implementation step does not need to buffer a failing diagnosis. The verify-pass rate of the trace lane is a direct reflection of diagnostic capability, rather than downstream brute-force recovery.

6.2 The Mechanism: Per-Call Information Density

The architectural resilience of the trace-augmented lane is driven by per-call information density. A single structured trace query effectively replaces a highly iterative, 15-call static grep walk.

We quantify this density by analyzing the exact tool-call vocabulary the agent converges upon when given access to the AppMap MCP surface. Across the 216 trace-augmented cells underlying our primary results (totaling 289 trace calls), the agent overwhelmingly abandoned exploratory search in favor of a single deterministic query (Table 2):

Table 2: Trace-augmented tool utilization distribution.

MCP Function	Total Invocations	% of Trace Calls	Usage Across Cells	Mean Calls per Using-Cell
`get_call_tree`	236	82%	216 / 216 (100%)	1.09
`find_calls`	51	18%	47 / 216 (22%)	1.09
`find_queries`	2	<1%	2 / 216 (1%)	1.00

The `get_call_tree` function is invoked in **100% of cells**, averaging just over one call per trajectory. One call returns the diagnosis-bearing frame sequence in its entirety, allowing the agent to immediately transition to the implementation phase. `find_calls` acts as a targeted follow-up in roughly one-in-five cells when the initial tree requires specific method resolution.

This narrow vocabulary is the mechanical reason tightening the budget to $b = 3$ barely moves the trace lane's performance curve. A static agent (`code-rca`) burns its initial budget enumerating file paths and inspecting imports, frequently exhausting its 10-call cap before it can read the actual implementation logic end-to-end. The 10-call cap is binding on `code-rca` because text search is sequential; it is functionally loose on `appmap-rca` because the trace allows each call to perform an order of magnitude more work.

6.3 Robustness of the Cost-Verify Trade Space

Figure 4 visualizes the cross-lane dynamic as a trade-off space: every cost saving achieved by the static baseline is paid for with a degradation in verification capability.

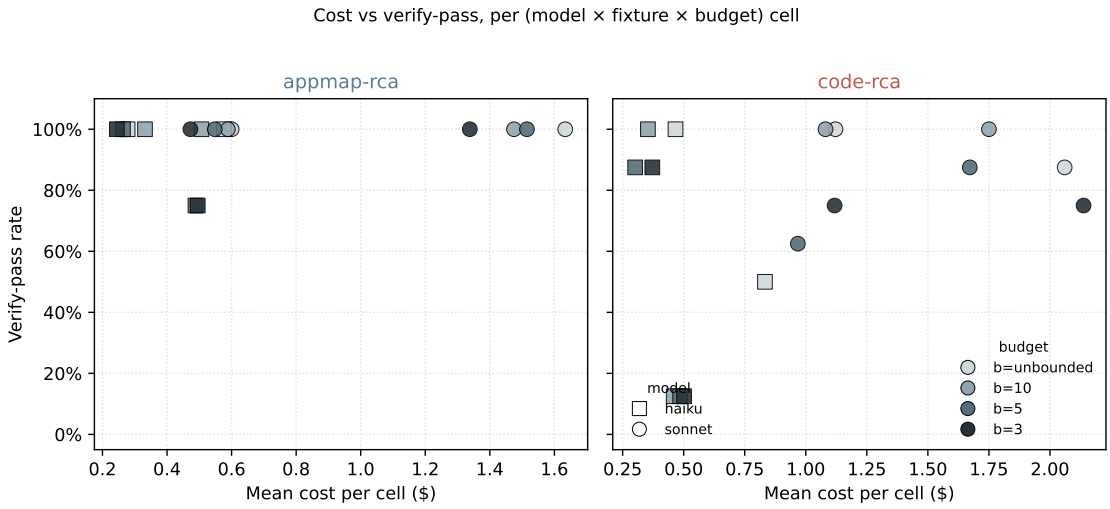


Figure 4: Cost vs verify-pass per (model × fixture × budget) cell, by lane. Points are colored by budget level (darker = tighter); markers distinguish models.

When the `code-rca` budget is tightened, the required API spend drops, but the cell clusters shift

cleanly downward on the verify-pass axis. There is no "free lunch" on the static baseline. Conversely, the `appmap-rca` clusters sit uniformly at the 100% verify-pass ceiling across the cost range. Tightening the trace budget yields a 20% reduction in token cost without sacrificing any accuracy, showing that trace augmentation is both cheaper and higher-fidelity than static exploration across all measured thresholds.

This collapse in static verification is not an artifact of a single pathological cell. As shown in Figure 5, every methodology-admitted cell experiences a `code-rca` capability drop as the budget tightens. While the exact slope of the collapse depends on the model's structural priors (e.g., Sonnet's extensive priors degrade more gently than Haiku's), the systemic failure of the static lane is universal.

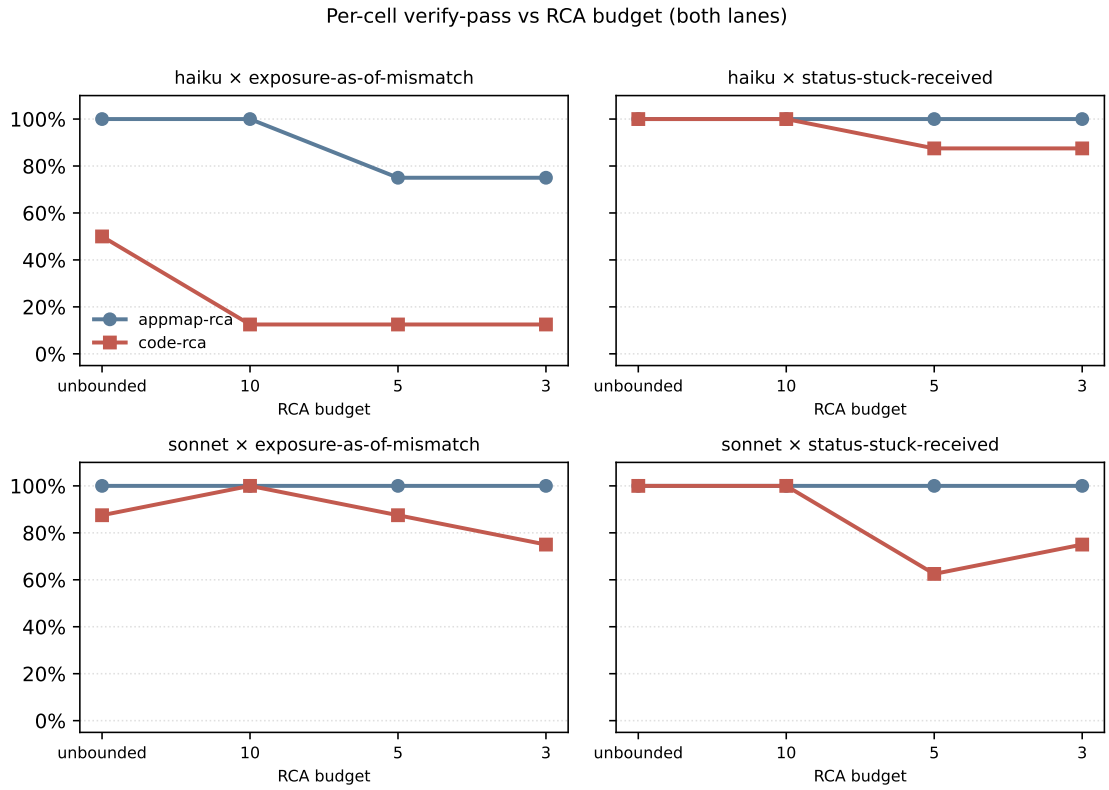


Figure 5: Per-cell verify-pass vs RCA budget (both lanes), small multiples per (model × fixture).

6.4 Boundary Conditions & Threats to Validity

The methodology and empirical findings of this study are subject to two specific boundary conditions regarding generalizability:

1. **Fixture Generalizability & Execution Limits:** While the two-fixture sweep identifies the core capability gap, our expanded 11-fixture suite (Section 5.2.1) demonstrates the limits of extreme cost optimization. A pure compact architecture (Haiku + trace at $b = 3$) fails on highly complex implementations, such as two-phase-commit rollbacks, that require the

structural reasoning of a frontier model during the implementation phase. The hybrid pipeline (Compact RCA + Frontier Fix) is required to generalize the high-verify results across a diverse enterprise suite.

2. **Model & Enforcement Generalizability:** Our evaluation is executed entirely within the Claude Code execution loop (v2.1.159) via hard-enforced `PreToolUse` hooks. Alternative agent architectures relying on soft, prompt-only budget enforcement (e.g., raw OpenAI API sessions) are known to exhibit high rates of instruction override and budget overshoots. Further study is required to verify if the trace-density effect scales identically across model families lacking strict runtime tool interception.

7. Conclusion & Future Work

As autonomous coding agents transition into mainstream enterprise environments, their efficacy is bottlenecked by the information density of their context-gathering tools. The prevailing reliance on iterative static analysis forces agents to consume vast token budgets on sequential file exploration, a brittle methodology that collapses under resource constraints.

This study demonstrates that providing agents with structured runtime telemetry alters the economic and capability frontiers of automated program repair. By isolating the root-cause analysis stage from downstream patch implementation, we established three primary conclusions:

1. **Budget Resilience:** While static-analysis diagnostic accuracy collapses to 28% under a strict three-call tool budget, trace-augmented agents maintain 100% diagnostic accuracy by collapsing a 15-call grep trajectory into a single, high-density runtime query.
2. **Elimination of Downstream Buffering:** Because trace-augmented agents reliably synthesize correct root-cause diagnostics, they eliminate the need for frontier models to waste input tokens re-verifying context during the patch-generation stage.
3. **The New Cost-Capability Frontier:** Trace augmentation enables compact, highly constrained models to match the diagnostic performance of unconstrained frontier models. Deploying a hybrid architecture (a budget-capped Haiku diagnostic paired with a Sonnet implementation step) recovers near-baseline verification pass rates across complex enterprise topologies while cutting financial expenditure in half.

Future Work: The Selectivity-Control Cohort

While this paper maps the cost-capability frontier on pristine, methodology-admitted fixtures, our immediate future work will address the boundary condition established in Section 3: *solution leakage*.

During our dataset admission pipeline, we explicitly filtered out bug reports that named the buggy mechanism or file path in the issue text, reserving them in a **selectivity-control cohort**. Our hypothesis states that when an issue description contains solution leakage, as is prevalent in benchmarks like SWE-bench, the diagnostic phase becomes degenerate. In future runs, we will execute

both the `code-rca` and `appmap-rca` lanes against this withheld cohort. We predict that the capability gap between static and dynamic evidence channels will collapse to zero on these leaked fixtures, which would show that pristine issue hygiene is a precondition for evaluating agentic root-cause analysis.

8. Data & Artifact Availability

Every result in this paper is produced by the project's benchmark harness and archived in the project repository. Each run-set lives under `reports/<run-set>/`: per-cell trajectories (`runs/<step>__<fixture>__…`), the run manifest (`runs.jsonl`), and the generated tables and figures the chapters cite. This appendix maps the named studies and derived views used above to their archived sources so the results can be reproduced; nothing in the body depends on reading a file listed here.

8.1 Run-sets

Cross-lane budget sweep (§5.1, §6) — the headline study: two admitted fixtures × two models (sonnet 4.6, haiku 4.5) × four RCA budgets (unbounded, 10, 5, 3) × N=8, 256 cells, integration-test source withheld from the RCA agent.

- Raw run-set: `2026-05-31-cross-lane-itwithheld-mini-sweep` (`runs.jsonl`, 256 rows, with `PROVENANCE.md`).
- Derived analysis bundle: `budget-sweep-2026-05-31-cross-lane-itwithheld` (`data.json`, `figures/`, `tables/`, `summary.md`) — the source of the §5 and §6 figures and the §5.2 / §6.5 tables.

11-fixture cost suite (§5.2.1) — the cost-claim generalization: `2026-06-02-cost-suite-3way`, a config-paired merge (per-fixture, never aggregated across configs) of three component run-sets on the identical 11-fixture admitted-or-priors-solvable set:

- Static baseline — sonnet × code-rca × unbounded (extracted from the methods upper-bound):
  `2026-06-02-sonnet-code-unlimited-suite`
- All-haiku candidate — haiku × appmap-rca × b=3:
  `2026-06-02-haiku-appmap-b3-suite`
- Hybrid pipeline — haiku b=3 RCA + sonnet fix:
  `2026-06-02-haiku-rca-sonnet-fix-hybrid`

The per-fixture side-by-side referenced in §5.2.1 is the merge's `tables/cost-3way-comparison.md`.

8.2 Generation scripts

The derived tables and figures are regenerated from each run-set's `data.json` / `runs.jsonl` by:

- `scripts/cost_frontier.py` — the §5.2 cost-frontier table.
- `scripts/report_budget_sweep.py` — the §6.5 per-row budget-vs-success table.
- `scripts/figures_budget_sweep.py` — the §5.1, §6.3, §6.4 figures.
- `papers/runtime-rca/_mcp_function_breakdown.py` — the §6.2 MCP-function invocation counts.
- `papers/runtime-rca/_render_tooling_figure.py` — Figure 1.

8.3 Supporting analyses

Three in-repo analysis memos record findings the body refers to by name:

- *fix-step-buffers-RCA-quality* (§5.2.1, §6.1) — the measurement that a capable fix step ships passing patches on a fraction of trials whose RCA is only partial: `papers/runtime-rca/findings/fix-step-buffers-rca-quality.md`
- *Gradle daemon-reap infrastructure bug* (§6.6) — the verify-flake source repaired before the §5 numbers were finalized, and the repair: `papers/runtime-rca/findings/reap-stop-kills-sibling-daemons.md`
- *RCA-resistance vs. fix-resistance dichotomy* (§6.4) — the priors-resistance axes the fixture pair was chosen to stress, from the companion methods paper: `papers/methods/findings/rca-resistance-vs-fix-resistance-dichotomy.md`

References

[1] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. Narasimhan. SWE-bench: Can Language Models Resolve Real-World GitHub Issues? In International Conference on Learning Representations (ICLR), 2024. arXiv:2310.06770.

[2] OpenAI. Introducing SWE-bench Verified. August 2024. https://openai.com/index/introducing-swe-bench-verified/.

[3] R. Aleithan, H. Xue, M. M. Mohajer, E. Nnorom, G. Uddin, and S. Wang. SWE-Bench+: Enhanced Coding Benchmark for LLMs. 2024. arXiv:2410.06992.

[4] S. Liang, S. Garg, and R. Zilouchian Moghaddam. The SWE-Bench Illusion: When State-of-the-Art LLMs Remember Instead of Reason. In Proceedings of the IEEE/ACM 48th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), 2026. arXiv:2506.12286.

