AUTONOMOUS ROOT CAUSE ANALYSIS WITH FRONTIER MODELS

WHAT IS RUNTIME CONTEXT? AND WHY IS IT MORE IMPORTANT THAN USING THE LATEST MODEL?

Mark Williamson, Nick Read, Finn Grimwood, Ed Tait, Marco Barisione - undo.io

undo

Background

Tokenmaxxing is gone, long live Valuemaxxing

A year ago, AI mandates rolled out across major tech companies: *just use AI, more is better*.

Today, commercial software development routinely uses **coding agents** - with many developers now reporting that most of their code is written through agents, not directly.

AI usage has since exploded, driving up spend on top models. **Focus is shifting from adoption to ROI**, with greater scrutiny on where AI spend goes and the value it delivers. Nobody wants to lose the edge AI tooling provides, but costs need to remain sustainable.

This is a balancing act! Engineering organisations must simultaneously:

Successful AI strategies need to balance these two, conflicting goals. Coarse organisational mandates like "prefer Sonnet to Opus" or "use no more than N tokens / week" minimises investment costs but risk the productivity returns that motivated AI adoption.

Typing code isn't the bottleneck on development

Meanwhile, it is widely acknowledged that writing code is a small fraction of development work. Delivering (and maintaining) a working product requires a huge effort in *understanding* software behaviour, in particular **root cause analysis (RCA)** of unexpected behaviour.

This is equally true for new code as for existing code that's been there for decades. **Without sufficient AI gains in understanding runtime behaviour, Root Cause Analysis will become a progressively-worse bottleneck on the Software Development Lifecycle.**

The LLM isn't the bottleneck on intelligence

A coding agent's reasoning comes from the LLM (Large Language Model) it uses. Major new LLM releases are now announced monthly or even more frequently, so raw reasoning isn't in short supply. And yet, the LLM's power is only as good as the **context** it can read. Without the right inputs even the smartest model will fail to give a good answer.

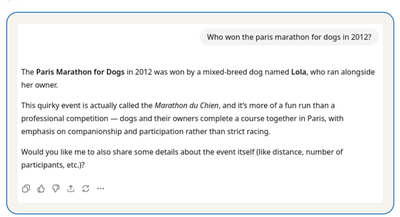


Fig 1: To the best of the authors' knowledge there is (sadly) no Paris Marathon for Dogs - without access to web search this agent is confidently wrong

in other words

AI = MODEL + CONTEXT

Our work

Runtime context

In discussing AI's ability to reason about software behaviour, we find it helpful to divide the information it uses into two broad categories:

Static context is always available in a correctly-configured system. The quality of **runtime context** available depends on many factors: the detail in a bug report, the fidelity of built-in logging, etc. **Process recordings**, as by **Undo**, rr and Microsoft's TTD are an ideal form of this, containing a precise and complete representation of program state during a particular execution.

Runtime context is the bottleneck on RCA

We've already seen that providing the correct context is essential for LLM reasoning. To illustate the power of runtime context in reasoning about software we'll consider what the most powerful frontier models can achieve when their context is limited.

We used our internal benchmark to test **Fable 5** on DuckDB issue #20086, which involved unexpected output from an SQL query. DuckDB's codebase is just ~0.5M lines of code.

	Static Context (just code and bug report)	Runtime Context (re-runs, logging, process recording)
Elapsed time	> 1 hour	12 - 24 minutes
Cost	$37.60	$2.88 to $6.31
Root cause	? (confidently wrong)	✓ (correct answer)

Note that the agent is always able to give a ***confident*** answer - they are very good at doing this! But when deprived of crucial information about the system's runtime behaviour, models are forced to speculate and provide a misleading results in return for the resources consumed.

Measuring the effectiveness of agentic RCA

We set out to build an objective measure of how well AI agents can perform **autonomous root cause analysis** (RCA) of **non-trivial bugs in C and C++ codebases**.

Fully autonomous RCA is the most challenging test available and can be measured objectively, without requiring a human in the loop providing feedback.

Non-trivial bugs are the ones that are most difficult to solve and least predictable to analyse. These are the ones that keep working software away from customers. These usually involve one (or more) of:

- Memory management
- Corrupt internal state
- Concurrency bugs
- Unexpected results
- Hangs

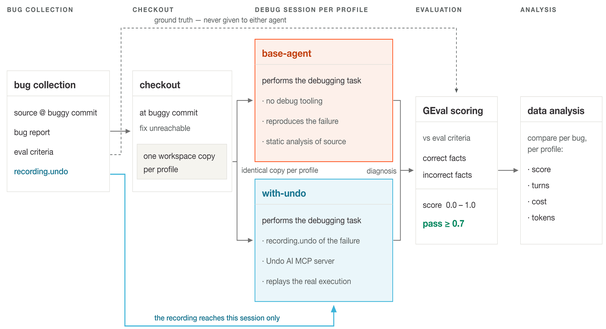
By their very nature, these bugs occur in **large codebases** with **complex invariants** around **complex state**. The root cause is typically far from the observed symptom, making it impossible to identify it from a simple backtrace or pre-defined logging.

Methodology

Bug library benchmark

Our goal is to test the ability of AI agents to reason about bugs involving complex state and control flow. Bugs are hand-selected to be specifically challenging for LLM reasoning and added to our "Bug Library" benchmark system.

Our goal is to simulate the process of resolving such a bug through reasoning about the code and its dynamic behaviour - the same challenge faced by engineers to investigate novel bugs "in the wild".



1. Bug collection — each bug is a real, historic defect, stored with its source, a symptoms-only report, ground-truth criteria and a recording of the failure

2. Checkout — the code is restored to just before the fix, with everything after made unreachable. This is to simulate a point in time of the codebase's history.

3. Debug session per profile — two agents get identical copies and the same task. The agents are given free reign; one with Undo AI and one without.

4. Evaluation — GEval used to evaluate presence of correct and incorrect information in returned summaries. More correct facts make for a higher score, with agents penalised for presence of known incorrect facts and red-herrings. Pass marks are >= 0.7.

5. Analysis — the two sides compared on score and on what it cost: turns, money, tokens

Do agents cheat at benchmarks?

AI agents are inventive, with a strong drive to provide an *answer to the question asked*. This makes them challenging to benchmark consistently, given their expertise in navigating computer systems and data.

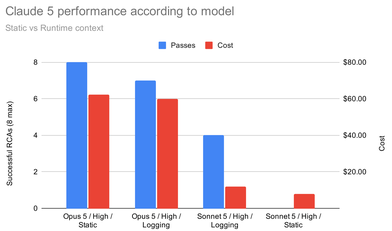
These skills can easily undermine the behaviour we want to measure, which is the ability to reason about and explain a program's observed behaviour.

- git history used as a shortcut - for a recently (or artificially) introduced bug the changeset history can make a bug obvious. Agents will examine the current branch, search for other available branches and even access known remotes to gain knowledge.
- Bug information in the training corpus - for Open Source codebases, bug tracker data from pre-training may allow the model to answer using its innate knowledge instead of the tools provided. This effect appears more noticeable in larger models.
- Internet access - even without git, models will attempt to access reference copies of source code (obtaining download tools, if required) to compare for injected bugs.

Findings

Model and runtime context

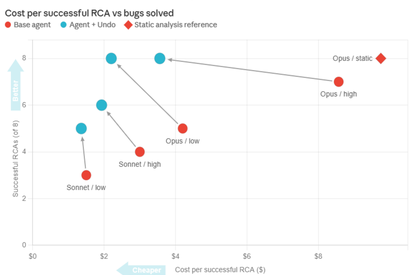
We benchmarked against closed-source bugs, to allow testing against recent models without their training from public data skewing our results. We used two current-generation models, Opus 5 (a model for coding, architecture and planning) and Sonnet 5 (a smaller, cheaper workhorse).
Here we compare the effect of pure **static context** vs **context from logging** and re-runs.



Opus is **significantly stronger at static analysis** than Sonnet and root causes issues in the (relatively small) codebases under test without needing logging / re-runs but the cost is high. For larger codebases we have seen its analysis is insufficient to solve bugs even with logging. Sonnet is **much cheaper but with limited effectiveness** with logs (and no success with only static context).

Thinking effort and type of runtime context

Here, we compare the influence of the **high vs low effort** and the use of **logs vs Undo-based process recordings**. The models are allowed to run and re-run the software under test and to construct directed test cases as part of their analysis if needed.



We make several interesting observations from this data:

- Opus at High effort, with logs for runtime context is very effective at resolving issues. **Adding Undo process recording retained this effectiveness, at lower cost**.
- Opus at Low is cheaper than High for either runtime context source. It is significantly less effective with logging, whilst **process recording retains effectiveness at even lower cost**.
- Sonnet on High reasoning is **cheaper but less effective** than Opus on Low.
- Sonnet with access to Undo is **cheaper and better** than Opus with only logs
- Our market experience suggests process recording enables successful RCA in significantly larger codebases.

To truly maximise return on investment from AI tools it is not enough to look at model and token costs - you must maximise the quality of the context provided