AGENTIC DEBUGGING

TIME TRAVEL DEBUGGING WITH AI

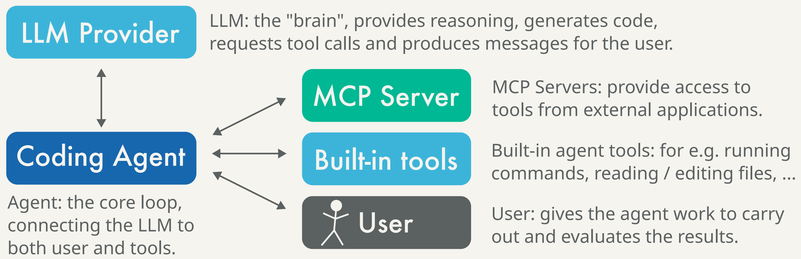
Mark Williamson, Finn Grimwood, Marco Barisione - undo.io

undo

Background

AI-augmented development

AI is changing how software is developed. The latest wave of LLM-based tools are **coding agents** that add autonomous capabilities to development tools.



Concerns and scepticism remain over this transition:

- Can LLM-based agents scale to large codebases?
- Can tendencies toward "Hallucination" (non-factual results) be tamed?

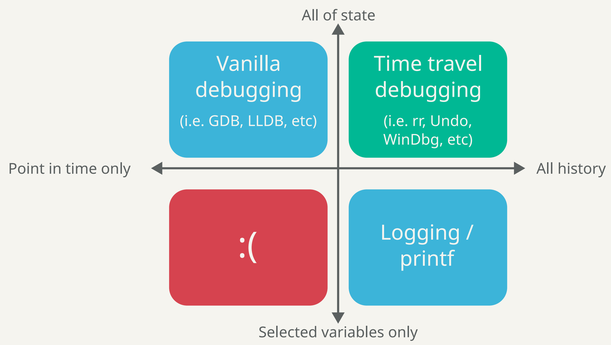
Most development isn't typing code!

Writing new code is a small fraction of development. Most technical effort is in understanding and debugging - this is equally true for new code as for existing code that's been there for decades.

Huge AI investments in generating code have not been matched by work in debugging and understanding it.

Time travel debugging

Time travel debugging provides a deterministic trace of a program's behaviour, precise to machine instruction granularity. These systems can wind time backwards and forwards to inspect all program state or query for properties such as "when was this function last called?" or "when was this memory corrupted?".



Time travel debuggers for C and C++ include `rr` [1] (OSS, Linux), Undo [2] (commercial, Linux) and WinDbg TTD [3] (closed source, free of charge, Windows).

Our work

Introducing agentic debugging

We have augmented agentic workflows with information about dynamic program behaviour. Prior work includes ChatDBG [5] (investigating LLM inspection of program state) and LDB [6] (using a debugger to fix errors during agentic coding).

Our work extends the state of the art by integrating the coding agent with a time travel debugger, to give:

- Complete visibility into program's dynamic behaviour - enabling **high quality context** for the LLM to understand what code is relevant.
- Deterministic replay - providing a verifiable **ground truth** about the system's behaviour, reducing the likelihood of hallucinations.

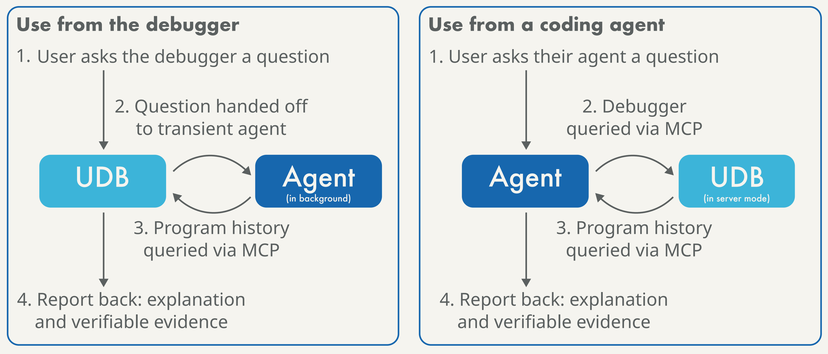
The explain command

We extended our time travel debugger (UDB) with a new command. The command works with an external coding agent to integrate with other tools and the LLM itself. The user can query recorded program behaviour in natural language, e.g:

```
explain what does the current backtrace mean?

explain what has gone wrong in this program?
```

The command returns a report to answer the user's question, including information from the program's execution history to justify its answer.



The extension is invoked within the debugger or used as a server to support a separate coding agent. The Model Context Protocol (MCP) provides the mechanism to expose time travel debug functionality to an agent.

Our implementation

Our implementation is built on the MCP Python SDK [4], integrated into the Python runtime environment of our UDB debugger. It supports Anthropic's Claude Code, Sourcegraph's Amp and OpenAI's Codex CLI as background agents (or any MCP-compatible agent when operating in server mode).

To view the BSD-licensed source code and additional supporting documents, scan the QR code provided.

Lessons learnt

Human tools vs AI tools

In principle, an LLM-based agent can work with any tool - in practice, tools designed for human users are not always a good fit. Take interactive debuggers:

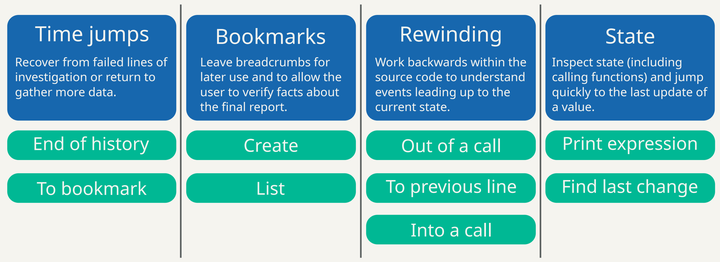
- Debuggers provide a broad set of composable features (e.g. breakpoints, watchpoints & stepping) to maximise flexibility for the user.
- Debuggers are extremely stateful, with the user expected to track what breakpoints are active and how different commands interact.

A debugger interface for an AI

LLM-based agents work best with a tool set designed for them. We've followed several principles for how to expose complex tooling to an agent:

- Supply a minimal set of tools for the task at hand - no distractions.
- Tools should be atomic and independent - no complex state to track.
- Work with the LLM - if it won't to use a tool as instructed, design a better tool.

The minimal tool set we identified for explaining program state is as follows:



By combining these, a coding agent can explain dynamic behaviour in real-world codebases such as GDB, with the results of one such bug investigation below:

Summary of Control Flow Analysis

Based on my investigation using UDB, I've identified the control flow that leads to GDB getting stuck:

1. **Command Execution**: The user executes `p (int)strlen("abc")`
2. **Nested Inferior Calls**: During evaluation, GDB needs to:
   - First, allocate space in the inferior (frame #5: `value_allocate_space_in_inferior`)
   - This triggers an inferior call via `call_function_by_hand`
   - During this call, at frame #2 (`run_inferior_call` line 808), the stdin handler is unregistered via `ui::unregister_file_handler`
3. **Key Problem**: The stdin file handler (fd=0) is deleted during the first inferior call setup but is never re-registered when the inferior call completes or fails.
4. **Result**: After the stdin handler is deleted:
   - GDB continues to wait for events in `gdb_wait_for_event`
   - Only fds 13, 9, and 7 remain registered (no fd 0 for stdin)
   - GDB can no longer receive input from the user
   - The program appears to hang indefinitely

Key Bookmarks Created:

- `stdin_deleted_during_inferior_call`: Where `delete_file_handler(fd=0)` is called
- `nested_inferior_calls_stdin_issue`: Shows the nested inferior call structure
- `gdb_waiting_for_event_stuck`: Where GDB is stuck waiting with no stdin handler

The root cause is that the cleanup/restoration of the stdin handler after an inferior call is not happening properly, likely due to the nested nature of the inferior calls or an exception/error path that bypasses the normal cleanup.

References

[1] https://rr-project.org
[2] https://undo.io
[3] https://learn.microsoft.com/en-us/windows-hardware/drivers/debuggercmds/time-travel-debugging-overview
[4] https://github.com/modelcontextprotocol/python-sdk
[5] ChatDBG: Augmenting Debugging with Large Language Models, Levin, Kyla H. and van Kempen, Nicolas and Berger, Emery D. and Freund, Stephen N. Proceedings of the ACM on Software Engineering Volume 2, 2025 - https://arxiv.org/abs/2403.16354
[6] Debug like a Human: A Large Language Model Debugger via Verifying Runtime Execution Step-by-step
Li Zhong and Zilong Wang and Jingbo Shang, 2024 - https://https://arxiv.org/abs/2402.16906