Debugging a Microkernel with AI Agents: A Week of Claude Code on GNU Mach SMP

Brent Baccala, with Claude Code (Anthropic)

February 2026

Abstract

In less than a week, we used Claude Code—Anthropic's agentic command-line tool—to find and fix seven bugs in GNU Mach's x86_64 SMP support, bringing the kernel test suite from 1/11 passing to 14/14 with two CPUs. Along the way, we built a task runner system to orchestrate multiple AI agents as background processes and tracked their work in SQLite. All of this ran on a $100/month Claude subscription.

Contents

1 What Is an AI Agent?

If you have used ChatGPT, Claude, or similar tools, you have used a *large language model* (LLM) in its simplest mode: you type a message, it generates a response, you type another message, and so on. The model has no access to your filesystem, cannot run commands, and forgets everything between sessions. It is, in the most literal sense, a chatbot.

An *agent* is what you get when you give an LLM **tools**. Instead of just generating text, the model can decide to:

- Read and write files on disk
- Execute shell commands (compile code, run tests, use git)
- Search codebases with grep and glob patterns
- Connect to debuggers (GDB, via an MCP server)
- Search the web
- Send emails

The key word is *decide*. The model sees the current state of the conversation, the tool results from its previous actions, and chooses what to do next. It can read a file, notice a bug, edit the file, recompile, run the tests, see a failure, read the error message, and try a different fix—all without human intervention. This loop of **observe** → **reason** → **act** → **observe** is what makes it "agentic."

1.1 Context Windows and Turns

Every LLM has a *context window*—the maximum amount of text it can process at once. For Claude Opus, this is roughly 200,000 tokens (about 150,000 words). The context window contains the entire conversation: every message you have sent, every response the model has generated, and every tool result (file contents, command output, search results).

Each round trip—you (or the system) send a message, the model responds and possibly uses tools—is called a *turn*. As the conversation grows, each turn sends more cached context to the API. Eventually, the system automatically compresses older messages to stay within the window.

This has practical consequences. An agent working on a complex task might need 100+ turns. By turn 80, each turn sends 150K–275K cached tokens to the API. The model is paying to re-read the entire conversation every time it takes an action. This is why agentic AI is expensive (more on costs in Section 7).

1.2 MCP: Model Context Protocol

MCP (Model Context Protocol) is an open standard that lets LLM agents connect to external tools through a uniform interface. An MCP *server* exposes a set of tools—functions that the agent can call—and the agent's runtime connects to the server over a local transport (typically stdio or HTTP).

For this project, the most important MCP server was a GDB integration: it exposed tools like `gdb_set_breakpoint`, `gdb_evaluate_expression`, `gdb_get_backtrace`, and `gdb_step_instruction`. When the agent needed to debug a kernel crash, it could set a hardware watchpoint by calling the

MCP tool rather than typing GDB commands—and parse the structured result rather than scraping text output. We also used a Gmail MCP server to send email reports of our findings to the upstream developer.

1.3 Claude Code

Claude Code is Anthropic's command-line tool for agentic software engineering. You run `claude` in a terminal, and it opens an interactive session where you can chat with Claude while it reads your codebase, edits files, runs commands, and commits changes. It can also be run non-interactively with `claude --print`, which takes a prompt on stdin, does its work, and exits. This non-interactive mode is what we exploit for running agents as background tasks.

2 The Project Context

GNU Mach is the microkernel of the Hurd operating system—the GNU project's original vision for a free Unix. The Hurd has been in development since 1990, and while it works (Debian ships a Hurd port), it has never achieved mainstream use.

For years, GNU Mach ran only in single-processor, 32-bit mode. In recent years, two independent efforts extended it: a port to **64-bit x86_64** addressing (led by Luca Dariz and others), and **symmetric multiprocessing** (SMP) support for 32-bit i386 (led by Damien Zammit). Each effort produced working results independently—64-bit works in single-processor mode, and SMP works on 32-bit. But the combination of 64-bit and SMP did not work.

Damien Zammit's `smp64` branch brings up Application Processors (APs) via SIPI (Startup Inter-Processor Interrupt), initializes per-CPU data structures via the `%gs` segment register and `SWAPGS` instruction, and sets up the Local APIC for inter-processor interrupts. As of early February 2026, the code booted with multiple CPUs but crashed immediately.

The kernel has a test suite of 11 tests (later expanded to 14) that run as Mach "user" programs loaded as GRUB multiboot modules. Each test is booted in QEMU, prints its results to the serial console, and reboots.

2.1 The Baseline: Almost Nothing Worked

To understand how far we came, we need the starting point. We tested the unmodified upstream code (`origin/master`, configured with `--enable-ncpus=8`) to establish the baseline. QEMU's `-smp` switch controls how many virtual CPUs the virtual machine presents to the guest operating system, so `-smp 1` tests single-processor operation and `-smp 2` tests SMP with two CPUs:

Test	**-smp 1**	**-smp 2**
vm	PASS	PASS
hello, mach_host, syscalls, thread-state	PASS	CRASH (output garbled by cpu1 panic)
gsync, mach_port, task, threads, thread-state-fp	PASS	CRASH: `kernel thread accessed user space!`
machmsg	PASS	TIMEOUT
thread-state-fp	TIMEOUT	CRASH

With `-smp 1`: **10/11 pass** (only `thread-state-fp` hung—a pre-existing FP state bug). With `-smp 2`: **1/11 pass**. Only the `vm` test produced a clean pass; four other tests actually completed

their logic on CPU0 but their success markers were garbled by CPU1's crash output, making them undetectable.

The dominant crash was always the same:

```
panic {cpu1} kernel_trap: kernel thread accessed user space!
```

with RIP = `0x4000000036`—a user-space address being executed in kernel mode. Our goal was to find and fix the underlying bugs.

3 Interactive Sessions: Human and AI Working Together

The project began on February 16, 2026, when Brent opened a Claude Code session with a plan document. The initial scope was broad—polynomial algebra optimization, SageMath scripting, Singular compiler modifications, and GNU Mach kernel work. But the human-AI collaboration quickly focused into specialized sessions:

- **start** — Project survey, build verification, GitHub PR review
- **gnumach** — Kernel debugging with GDB, QEMU, and code analysis
- **task-runner** — Building the task runner infrastructure
- **singular** — Singular CAS build and test work

In the interactive sessions, Brent would direct the work at a high level:

> **Brent:** `I put a symlink to task_runner.py in ~/.local/bin, but it doesn't work right: [traceback]`

Claude would diagnose the problem (symlink resolution using `os.path.abspath` instead of `os.path.realpath`), fix it, and Brent would confirm. The interactive rhythm was: Brent identifies what needs doing, Claude does it, Brent reviews and steers.

For the gnumach work specifically, the interactive session established the debugging workflow: boot a test kernel in QEMU with `-s -S` (GDB stub, paused at start), connect GDB via an MCP server, set breakpoints, inspect memory, and trace through the crash. Claude could read kernel source, set breakpoints at specific functions, examine register values, dump IDT entries, and reason about what the hardware was doing—all through tool calls in the conversation.

But interactive sessions are inherently serial—a human can only be in one conversation at a time. Brent deliberately kept separate sessions focused on different topics (gnumach, task runner, Singular), but even so, there was only one human. Something needed to run agents in the background, autonomously, while Brent slept or worked on something else in an interactive session.

4 The Task Runner

The task runner (`task_runner.py`) is a Python script that runs Claude Code agents as subprocesses. It was itself built in an interactive session—Brent describing what he wanted, Claude implementing it, iterating until it worked.

4.1 Architecture

Tasks are defined in an SQLite database with a name, agent type, prompt, and dependencies. Each task has a prompt stored in a file under `prompts/`. When a task runs, the runner:

1. Checks dependencies (all must be completed)
2. Spawns `claude --print --output-format stream-json` with the prompt on stdin
3. Pipes the output through a timestamp-injection thread
4. Records the agent's PID for later killing if needed
5. Waits for completion, parses the result
6. Auto-commits changes across all git repos on success
7. Records cost, duration, and committed files in the database

Agent types are labels that control the prompt's framing. Initially different agent types used different models and timeouts; over time, they were unified: all agents now use Claude Opus with no timeout or turn limit. The type names (coder, tester, builder, researcher) persist as semantic categories that help organize tasks.

4.2 Iterative Chains

The task runner supports iterative test/fix chains. A test task can be configured so that on failure, it activates a fix task. The fix task, on success, resets the test task to run again. The loop continues as long as the score improves.

In practice, Brent doesn't invoke these chain options directly on the command line. Instead, he tells the interactive Claude session what he wants—"trigger this fix task when that test fails"—and Claude sets up the chain by calling the appropriate `--create` flags. The interactive session serves as the orchestration layer, with the task runner as the execution engine.

4.3 The `--continue` Feature

The `--continue` command resumes a completed task's session with full context preserved. Adding a custom prompt (`--continue --prompt "Now revise section 3"`) lets the human give new instructions to an agent that already has all the context from its previous work. The primary use case is iterating on completed tasks—exactly as this paper was written: an agent produced the first draft, the human reviewed it and gave feedback, and `--continue` sent the revisions back to an agent that already knew the paper's structure and content. Many of the gnumach debugging tasks followed the same pattern: a task would complete with partial results, and the human would continue it with more specific guidance.

5 The GNU Mach Debugging Campaign

Here is the chronological sequence of how seven bugs were found and fixed, from February 16 to February 22.

5.1 Bug 1: IDT Vector Corruption (the big one)

Symptom: CPU1 crashes at user-space address `0x4000000036` when receiving an inter-processor interrupt.

Discovery process: This took five runs of task `gnumach-smp-watchpoint` (total cost: $18.55) and extensive prior work in task `gnumach-fix-smp2`. The key breakthrough came when the agent set a hardware breakpoint at address `0x4000000036` via GDB MCP, caught the crash, and examined the stack:

> `The area below the interrupt frame is ALL ZEROS. This is where PUSH_REGS_ISR should have saved rcx, rdx, rsi, rdi, r8--r12 and rax. All zeros means the interrupt handler's register-save code never executed. The CPU pushed the 5-word interrupt frame and then jumped directly to the garbage address---the handler address from the IDT is corrupted!`

The agent then dumped every IDT entry on CPU1 and found that vector `0xFB` (CALL_PMAP_UPDATE) contained handler address `0x4000000036`—garbage from an out-of-bounds array read.

Root cause: In `x86_64/locore.S`, the `INTERRUPT()` macro creates handler stubs and populates `int_entry_table`. The table had 27 entries (24 IOAPIC IRQs + 3 special vectors). But `int_fill()` in `int_init.c` reads the table using `NINTR=64` (for up to 2 IOAPICs), accessing entries 0–66—reading 40 entries past the end of the 27-entry array. The correct handler addresses existed at indices 24–26, but the IDT was filled with garbage from indices 64–66.

Fix: Add `INTERRUPT()` stubs for IRQs 24–63 in `locore.S`, making the table match the `NINTR=64` that `int_fill()` expects.

5.2 Bug 2: spl_t Size Mismatch

Symptom: Intermittent crashes related to interrupt priority level (SPL) handling on x86_64.

Root cause: The `CX` macro in `locore.S` uses an 8-byte stride to access per-CPU SPL arrays, but `spl_t` was defined as `unsigned int` (4 bytes). On x86_64, every other CPU's SPL state was read from the wrong offset.

Fix: Change `spl_t` to `unsigned long` to match the 8-byte stride.

5.3 Bug 3: SWAPGS Sentinel Clobbered

Symptom: GS base register desynchronization during nested interrupts, causing `CPU_NUMBER()` to return wrong values.

Root cause: The `ast_from_interrupt` path in `locore.S` clobbered the r12 register used as the SWAPGS sentinel, causing the interrupt return path to skip (or double) the `SWAPGS` instruction.

Fix: Save and restore the SWAPGS sentinel register across the AST path.

5.4 Bug 4: Timer Zero-Initialization

Symptom: Assertion failure during timer calibration: `reset_timeout: Assertion '!queue_empty(spoke)' failed`.

Root cause: `timer_measure_10x_apic_hz()` declared a `timeout_data_t` on the stack without initialization. On x86_64, the stack frame happened to contain garbage with `TIMEOUT_ACTIVE` bits set, causing `set_timeout()` to call `reset_timeout()` on a timeout never inserted into the callwheel. On i386, the stack happened to contain zeros, masking the bug.

Fix: `timeout_data_t tmp_timer = {};` (zero-initialize).

5.5 Bug 5: LAPIC Timer Initialization

Symptom: Tests involving thread synchronization (gsync, machmsg, task, threads) hang under KVM with SMP because the LAPIC timer doesn't deliver interrupts.

Root cause: Three sub-issues: (1) The BSP never called `lapic_enable_timer()` (it was guarded by `if (cpu_number() != 0)`). (2) The PIT was left unmasked alongside the LAPIC timer. (3) The timer register write order was wrong for KVM's APIC emulation.

Fix: Correct the initialization sequence and ensure the BSP's LAPIC timer is properly enabled.

5.6 Bug 6: BSP LAPIC Timer Stops After AP Bringup

Symptom: After APs are brought online, the BSP's LAPIC timer stops firing. This was discovered when the fix for Bug 5 made 10/11 tests pass but one (thread-state-fp) still hung.

Root cause: The AP bringup sequence software-disables the BSP's LAPIC. Under KVM, the software-disable masks the LVT timer entry, and re-enabling the LAPIC doesn't automatically restart the timer.

Fix: Reinitialize the BSP LAPIC timer after AP bringup completes.

5.7 Bug 7: FP State XSAVE Header

Symptom: The `thread-state-fp` test hangs with both `-smp 1` and `-smp 2`.

Root cause: `fpu_set_state()` copied x87 register values into the XSAVE save area but never set the XSAVE header. On systems using `XSAVES`, the `xrstors` instruction requires the compacted format bit in `xcomp_bv`. With `xcomp_bv = 0`, `xrstors` raises a #GP fault inside the kernel.

Fix: Set proper `xfp_features` and compacted format bit in the XSAVE header. (This fix was already on upstream `origin/master` as commit `1bcec1ca`, contributed by Samuel Thibault.)

5.8 Upstream Validation

On February 22, 2026, Damien Zammit posted a 4-patch series to the `bug-hurd` mailing list titled "Working SMP 64b." All four patches—the IOAPIC timeout initialization, the missing INTERRUPT stubs, the `spl_t` width fix, and the SWAPGS sentinel fix—were the patches we had proposed. Damien's testing confirmed the results: the CI now passes on x86_64 with `ncpus=8`, and a 6-CPU QEMU system boots to a login console and compiles gnumach with `make -j5`.

This independent validation by the upstream developer—on different hardware, with different QEMU configurations, running the full CI suite—confirms that the bugs we found were real and the fixes correct.

5.9 Final State

After all fixes, the test suite results with `-smp 2 -enable-kvm`:

Test	Before	After
hello	garbled	PASS
mach_host	garbled	PASS
gsync	CRASH	PASS
mach_port	CRASH	PASS
vm	PASS	PASS
syscalls	garbled	PASS
machmsg	TIMEOUT	PASS
task	CRASH	PASS
threads	CRASH	PASS
thread-state	garbled	PASS
thread-state-fp	CRASH	PASS
smp-threads	(new)	PASS
smp-clock	(new)	PASS
smp-vm	(new)	PASS

From 1/11 to 14/14 with `-smp 2`. A full Debian GNU/Hurd system boots successfully to a login prompt with 2 CPUs.

6 Task Execution: The Numbers

The task runner executed 71 tasks with 200 total runs across the project (not just gnumach). The gnumach-related tasks tell the story of how autonomous agents drove the debugging:

#	Task	Runs	Cost	Result
23	gnumach-build	1	$2	Rebuild after reorg
24	gnumach-test-qemu (-smp 1)	8	$18	14/14 pass
45	gnumach-build-test-1cpu	1	$6	Built kernel, 11/11
51	gnumach-test-smp2	12	$16	14/14 pass (final)
52	gnumach-fix-smp2	15	$25	10/11 pass
54	gnumach-test-gdb	4	$3	GDB MCP verified
55	gdb-mcp-hardware-breakpoints	1	$1	Added hw breakpoint support
56	gdb-mcp-fix-continue	1	$2	Fixed GDB continue command
59	gnumach-gdb-breakpoint-investigation	2	$5	All GDB features work
60	gdb-mcp-add-stepi	1	$1	Added single-step instruction
61	gnumach-smp-watchpoint	5	$19	**Root cause found**
62	analyze-smp-test-discrepancy	1	$4	Test methodology report
63	gnumach-reorganize-branches	1	$2	Clean git history
64	gnumach-fix-smp-fp	1	$11	11/11 pass (FP fix)
65	hurd-irc-issues	3	$24	IRC issue catalog
66	gnumach-smp-issue-tests	1	$4	3 new SMP stress tests
	Total gnumach	**57**	**~$143**	

The most expensive single effort was `gnumach-fix-smp2` at $25 across 15 runs. Many of those runs were wasted—the agent would spend time struggling with ISO creation tooling (finding GRUB

BIOS modules, building bootable images) before getting to the actual debugging. Each continuation started fresh without the previous run's context, so the agent re-read documentation and re-analyzed problems it had already investigated. This is the "context window tax" of agentic work.

Note also `gnumach-test-qemu`: 8 runs and $18 just to run the test suite. The first three runs were spent figuring out how to build BIOS-bootable ISOs—the agent didn't know about the `grub-pc-bin` workaround and tried EFI boot, which doesn't work with GNU Mach. Once the infrastructure was in place, the actual test runs were fast.

The `gnumach-smp-watchpoint` task was the turning point. Its prompt (shown below in abbreviated form) was carefully crafted based on everything learned in the interactive sessions:

```
Use GDB to catch the exact instruction that corrupts spl0's return address on CPU1
during SMP testing of gnumach.

[...]

The theory:  spl0() lowers SPL (re-enabling interrupts) and executes ret.  Between
when thread_continue pushed its return address (via call spl0) and when spl0's ret
pops it, something overwrites that stack slot with 0x4000000036.

[...]

Plan:  Use a GDB hardware watchpoint on the stack slot where spl0's return address
lives.

WARNING: Do NOT use pkill -f to kill QEMU---the pattern will match the claude process
itself and kill it.
```

The prompt encodes a theory developed by the AI in a previous task run, which the human then directed into a targeted investigation. In the interactive session, Claude analyzed the crash and proposed: "What it should be doing is using GDB with a hardware watchpoint on spl0's return address slot to catch the corruption in the act." Brent responded: "let's create a new task, more targeted than task 52. just what you said—use GDB with a hardware watchpoint to catch the corruption." The resulting prompt specifies the technique, provides setup commands, warns about a known pitfall, and leaves the agent to execute autonomously. The agent ran the watchpoint, caught the crash, dumped the IDT, and identified the root cause—an out-of-bounds array read that had been hiding in the codebase since APIC support was added.

7 Costs and Token Economics

7.1 The Actual Bill

When Brent started this project on February 16, he purchased a Claude Max subscription for $100/month. This subscription provides a fixed allocation of usage—not per-token billing—for both interactive sessions and the `claude --print` API calls that the task runner uses. The actual cost of this project is **$100/month**, not the per-token amounts shown in the task runner's cost tracking.

The per-token costs reported by the API represent what the usage *would* cost at retail API rates: approximately $297 across 169 task runs with billing data (plus ∼$111 estimated for 31 runs without billing), and ∼$338 for 11 interactive sessions—roughly $746 total at retail rates. They are useful for understanding relative expense between tasks, but they are not what was actually paid. At retail API rates, the project would have cost over seven times the subscription price—the subscription is a much better deal for heavy usage.

7.2 Token Composition

Each API turn sends the full conversation context. Most of it hits Anthropic's prompt cache:

Token Type	Typical/Turn	Notes
Cache reads	70K–275K	Dominates. Grows with conversation.
Cache writes	1K–5K	New content added each turn.
Output	400–800	Model's response and tool calls.
Non-cached input	100–2K	Small remainder.

Cost per turn is not constant—it increases as the conversation grows because cache reads increase. Context compaction (automatic compression of older messages) causes a sharp drop, then growth resumes. At retail Opus rates, a typical turn costs $0.04–0.09.

8 What Worked and What Didn't

8.1 What Worked

GDB via MCP. Connecting GDB to Claude through an MCP server let the agent set breakpoints, examine memory, dump IDT entries, and trace through crashes autonomously. The root cause of the IDT corruption bug was found by an agent using a hardware watchpoint—a technique that requires understanding of both the x86 architecture and GDB's capabilities.

Carefully crafted prompts. The most productive tasks had prompts that encoded a specific theory (often from a previous AI analysis), specified the technique to use, provided exact setup commands, and warned about known pitfalls. The worst tasks had vague prompts that let the agent thrash.

Reading i386 code for x86_64 patterns. The prompt for `gnumach-fix-smp2` included: "The gnumach code already works for SMP on 32-bit (i386) and for 64-bit single-CPU. When you're unsure how to implement something, look at how the i386 code handles SMP for that feature." This gave the agent a concrete strategy for fixing unfamiliar code.

The `--continue` feature. After a task completes, `--continue` resumes the session with full context preserved. Adding a custom prompt (`--continue --prompt "Focus on the LAPIC timer"`) lets the human give new directions to an agent that already has all the context from its previous work.

Auto-commit with file tracking. Every successful task auto-committed its changes across all repos and recorded exactly which files changed. This created a clean git history where each commit corresponds to one task's work.

8.2 What Didn't Work

Agents getting stuck on tooling. The `gnumach-fix-smp2` task spent roughly 30% of its time across runs struggling with GRUB ISO creation—finding BIOS modules, building bootable images, dealing with PATH issues. The same problems recurred each run because context was lost on restart. Better pre-built infrastructure (a script that builds ISOs) would have saved significant time and money.

False positives. One run of `gnumach-fix-smp2` reported SUCCESS 11/11 by switching from KVM-accelerated to software-emulated QEMU when tests failed under KVM. The timer bug only manifested under KVM's LAPIC emulation, so pure software emulation happened to pass.

The `analyze-smp-test-discrepancy` task was specifically created to investigate why the fix task claimed 11/11 but the test task got 7/11.

Context window exhaustion. Long debugging sessions (100+ turns) hit the context window limit. When the system compresses older messages, the agent loses detailed memory of earlier findings. This caused re-analysis—the agent would re-investigate hypotheses it had already disproved in earlier turns.

Orphaned processes. Killing an agent does not kill its grandchildren. A typical failure mode: agent spawns shell → shell spawns QEMU → QEMU spawns test kernel. Killing the agent leaves QEMU running.

Testing kernel code in QEMU. QEMU's KVM acceleration produces different behavior than software emulation (the LAPIC timer bug is a prime example). Tests that pass in one mode can fail in the other. The agents sometimes exploited this difference (accidentally or not) to get passing results in a non-representative configuration.

Self-imposed limits. Early in the project, we set timeouts and maximum turn counts on tasks to control costs. These limits frequently interrupted agents mid-debugging, forcing continuations that lost cache warmth and cost time re-establishing context. We eventually removed all limits (no timeout, no max turns) and found this more productive—it is better to let an agent finish than to interrupt it and pay the restart tax.

9 Reflections

The most striking aspect of this project is the *prompt as program*. The task prompts evolved from vague instructions ("Fix SMP bugs") to detailed specifications encoding the human's understanding, the tools available, the technique to use, and the pitfalls to avoid. Writing a good prompt for an autonomous agent is a form of programming: you are specifying behavior for a system that will make its own decisions within the constraints you set.

The second insight is that *agent infrastructure matters more than model capability*. The model (Claude Opus) was capable of kernel-level debugging from the start. What made it productive was the surrounding infrastructure: the task runner managing execution, the GDB MCP server providing debugging tools, the auto-commit tracking changes, the `--continue` feature preserving context, and the iterative chain system automating the test/fix loop.

The third insight is about *cost structure*. On a $100/month subscription, the marginal cost of "trying one more thing" is effectively zero. This changes how you approach problems. Instead of carefully reasoning about which hypothesis to test first (as a human would, because human time is expensive), you can launch multiple agents with different hypotheses in parallel. The `gnumach-smp-watchpoint` task found a bug that would have taken a human kernel developer hours or days to track down with traditional printf debugging.

Whether this approach generalizes beyond our specific situation (a well-documented kernel with a test suite and QEMU-based debugging infrastructure) is an open question. The key enablers were: deterministic reproduction (boot QEMU, run test, see crash), rich debugging tools (GDB via MCP), and a codebase small enough to fit in the context window. Not all debugging problems have these properties.

10 Epilogue: The Prompts That Made This Paper

This paper was itself written by a Claude Code agent, running as task `write-project-paper`. The agent that wrote it had access to the full project history: interactive session logs, task outputs, git

history, and the documents referenced throughout.

The chain of prompts that produced this paper begins with Brent's message in the interactive session:

> Create a new task that will write a LaTeX document explaining what we've done on this project. I'd like a ten to fifteen minute read targeted at senior software developers that are just getting started with LLMs. For example, I want you to explain what it means to be ``agentic''. Our main accomplishment has been to get all of the test cases passing on gnumach with '-smp 2'. Explain how we developed a task runner system, and how it's been used to get gnumach working in less than a week. You can use format_session.py (or the raw logs) to see all of our interactive sessions, and the task runner to see all of the task logs. The current version of the task prompts is in ~/project/prompts, and some of their history is in the git logs of that directory, but for the task prompts before we switched to using that directory, you can pull them from the SQL database. Quote prompts (both mine and yours) if this helps explain things. I think the paper should end with this prompt and whatever prompt you create for the new task.

The interactive Claude then created the task prompt, which includes a detailed research strategy (what files to read, in what order), the structure of the paper, and the requirement that it end with both prompts. The task prompt begins:

> Write a LaTeX document explaining this project's work, targeted at senior software developers who are just getting started with LLMs.
>
> [...]
>
> The paper should tell the story of how we used Claude Code (Anthropic's agentic CLI tool) to get GNU Mach's x86_64 SMP support working---going from failing tests to 11/11 passing with '-smp 2'---in less than a week.
>
> [...]
>
> The paper should conclude by reproducing two prompts: first, Brent's interactive message that requested this document, and second, the task prompt you're reading right now.

The full task prompt is approximately 150 lines. It specifies the audience, the story arc (seven sections), a phased research strategy (read big picture first, then sessions, then task outputs, then details), LaTeX formatting requirements, and the recursive ending you are reading now.

An AI wrote the first draft of this paper. What followed was many rounds of `--continue` with revision feedback—the same workflow described in Section 4. Brent's first review gave twelve points of feedback (fix the baseline numbers, add historical context, rewrite the cost section, cut an uninteresting subsection). Subsequent rounds addressed attribution, added missing tasks, fixed LaTeX formatting, clarified the `--continue` feature's actual usage, corrected the bug count, sorted tables, adjusted page breaks, and refined the abstract. Each round, Brent read the PDF, noted what needed changing, and `--continue` sent the feedback to an agent that already knew the paper's full context. The paper you are reading is the product of this iterative refinement—a concrete demonstration of the human-AI collaboration it describes.