Case Study: ChatGPT 5.6 Sol successfully performs an "impossible" refactor of 160 files and outperforms a 10x developer.

Date : July 14, 2026

Author : Joël Abenhaïm, AI Software Engineer @ AI Sovereign Labs

Contact : joel@aisovereignlabs.ai

→ Version française de ce document

Summary :

A state-of-the-art AI coding agent, AICode, powered by the OpenAI ChatGPT 5.6 Sol model in "MAX reflection" mode, was used to undertake a massive, extremely difficult refactoring task on a complex AI software system. The codebase comprised 717,725 lines of TypeScript and 3,648 files, featuring a custom orchestrator backend, a custom React frontend, and a custom RPC data bus connecting the two.

The goal was to make the chatbot’s streaming window closable without interrupting execution, while ensuring the stream could be resumed if the window was reopened while the process was still running. This operation violated a fundamental architectural invariant: the guarantee that a window would open at the start of an AI request and remain open for its entire duration.

The task was exceptionally difficult, requiring an end-to-end reconstruction of a large, complex application's core while dismantling a central invariant. It involved navigating numerous challenges regarding cross-dependencies, execution sequencing, cache restoration, memory leaks, and race conditions. Even for a "10x developer," I would consider this task unfeasible, regardless of the time allocated. It is the sort of job that usually necessitates rewriting components from scratch, a far more realistic approach than refactoring in such a scenario.

Yet, the AI coding agent based on ChatGPT 5.6 Sol successfully completed the operation in just three days with minimal developer supervision. It modified hundreds of files and simultaneously

adhered to dozens of rules and invariants while maintaining architectural integrity. The result was the modified software exactly as intended: fully functional, with full test coverage, with zero visible bugs and zero regressions upon the very first manual test following the refactor. No human code review was performed, as it was unnecessary; furthermore, the nature and scale of the problem would have made such a review impossible.

This result relies on a probabilistic convergence method: 14 specification-tightening cycles and 17 automated code review cycles automatically eliminated 201 bugs and architectural defects prior to delivery.

Something that seemed unfeasible became feasible because AI surpassed human intelligence and was guided by a rigorous scientific method. This document explains precisely how the operation was conducted and provides the complete raw execution logs as evidence (in French language). These logs can be analyzed by an AI to verify their content and consistency.

Disclaimers:

This case study is NOT:

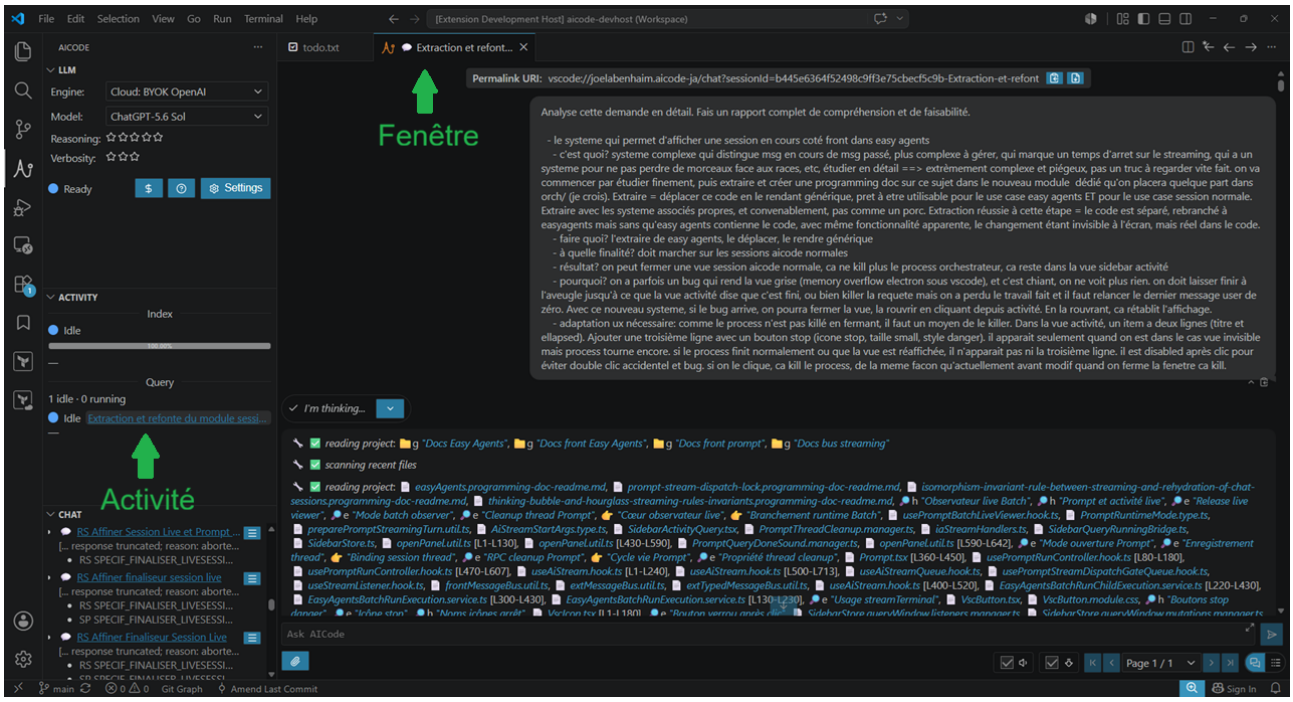
- a project written from scratch
- yet another Rust transpilation
- a clone of an open-source project found in the LLM's training data
- a Super Mario clone in HTML

It IS:

- a complex application maintenance task
- a unique problem with no open-source equivalent, to the best of my knowledge
- a genuine challenge, one of such magnitude that a "10x developer" could not handle the request without rewriting all or part of the program to alter such an axiomatic invariant
- a true engineering feat, 90% driven by ChatGPT 5.6 Sol Reasoning MAX
- 201 bugs, ambiguities, and architectural flaws automatically fixed by the AI agent, as opposed to a typical implementation using Claude Code, Codex, Copilot, or Cursor
- proof that one can program perfectly within complex legacy software without any need for human code review

1. Job description

The diagram shows two areas of the software: the Activity area and the tab where the window closes.



The goal is to allow the window to be closed without automatically interrupting the ongoing AI request—letting the request continue to run instead. Users should be able to stop the request via a "Stop" button in the activity view and reopen the window at any time while maintaining display consistency, just as if it had never been closed.

We want to decouple this functionality from the EasyAgents subroutine, where it is currently embedded within the existing code, turn it into a generic library, and refactor the main program to implement the chatbot window's open/close capability.

Resuming an active stream is challenging because the window's content is not simple text; it mixes response text, reasoning traces, tool calls, code blocks, and other UI elements. Reconstructing this display after closing the window requires locating and reassembling all these components.

Adding to this complexity are race conditions during closure, process interruptions while generation is underway, and edge cases regarding the display. The token stream becomes caught between three competing factors: the window opening, the retrieval of previously generated content, and the live streaming of new tokens.

2. Let's assign the task to the AI coding agent.

The request is submitted to the agent as follows:

> “ Analyze this request in detail. Produce a comprehensive report covering understanding and feasibility.
>
> - The system that streams an active session on the frontend within Easy Agents:
>
> - What is it? A complex system that distinguishes between current and past messages; it is tricky to manage, involves handling streaming pauses, includes mechanisms to prevent data loss due to race conditions, etc. It requires detailed study, it is extremely complex and full of pitfalls, not something to be looked at casually. We will start by analyzing it closely, then extract the code and create technical documentation for it within a new dedicated module (likely located somewhere under `orch/`). "Extract" means moving the code while making it generic, ready for use in both the Easy Agents use case and the standard session use case. The extraction must be done properly, including all associated systems, no sloppy coding. A successful extraction at this stage means the code is separated and reconnected to Easy Agents without Easy Agents actually containing the code itself; the visible functionality remains the same (the change is invisible on screen but real in the codebase).
>
> - What needs to be done? Extract it from Easy Agents, move it, and make it generic.
>
> - What is the goal? It must work with standard AI-code sessions.
>
> - What is the result? Closing a standard AI-code session view no longer kills the orchestrator process; the session remains visible in the activity sidebar.
>
> - Why? We sometimes encounter a bug that turns the view gray (an Electron memory overflow issue in VS Code), which is annoying because nothing is visible. Currently, we have to let it finish blindly until the activity view indicates completion, or kill the request—but that means losing the work done and having to restart the last user message from scratch. With this new system, if the bug occurs, we can close the view and reopen it by clicking on it in the activity list. Reopening it restores the display.
>
> - UX adaptation required: Since the process isn't killed when the view closes, we need a way to kill it. In the activity view, each item currently has two lines (title and elapsed time). Add a third line containing a stop button (stop icon, small size, danger style). This button should appear only when the view is hidden but the process is still running; it (and the third line) should not appear if the process finishes normally or if the view is displayed again. The button becomes disabled after being clicked to prevent accidental double-clicks and bugs. Clicking it kills the process, just as closing the window currently does. ”

3. Phase 1: Extraction of the liveSession library

Based on this prompt and a scoping response from me, the agent produced an initial specification for extracting a reusable library named `liveSession`, verified it multiple times, wrote the code, and verified it again. This refactoring effort involving 99 files, referred to as "Slice 1", is not detailed in this case study, as "Slice 2" is sufficient to understand the most challenging aspect. We begin our analysis of operations and logs starting from the beginning of Slice 2.

4. Phase 2: Reopening chatbot windows during streaming

The subsequent work was carried out without reviewing the Phase 1 code and without even manually running the program. All operations were conducted end-to-end, pure coding, over a period of three days before the program was executed a single time.

4.1. Creation of the specification

To kick off Phase 2, once the `liveSession` library has been extracted and is available to the chatbot windows, I return to the initial session with the agent, right where I had previously requested a specification. I clone the session and inform the agent that the Phase 1 extraction was completed in a separate session and that we are picking up from there for Phase 2. I then ask it to generate a specification. It creates an initial specification that modifies or creates 110 files across the shared, backend, frontend, and RPC middleware bus components.

Attached session log: liveSession-logs-phase2-ideation.pdf (83 pages)

4.2. Specification refinement cycles

The term "refining a specification" refers to the process of having the AI agent re-analyze a specification versus the source code, to check for errors or defects and correct the specification before programming begins. With each refinement, the AI agent provides a list of corrections and rewrites the corrected specification.

The specification underwent 14 refinement cycles, resulting in approximately 85 corrections. The table below provides the full AI agent session logs for these refinement operations as an attachment. Each operation represents a separate AI agent session, taking an average of 35 minutes to complete.

Note: Due to a migration bug associated with the ChatGPT 5.6 update, the list of corrections was unavailable for some operations; in these cases, the table estimates the count at 5.

Refinement Table:

# Refine	Number of fixes	Number of files affected	Attached session log
1	≈ 5	120	liveSession-logs-phase2-refine-01.pdf (19 pages)
2	≈ 5	122	liveSession-logs-phase2-refine-02.pdf (24 pages)
3	5	122	liveSession-logs-phase2-refine-03.pdf (25 pages)
4	4	122	liveSession-logs-phase2-refine-04.pdf (19 pages)
5	6	122	liveSession-logs-phase2-refine-05.pdf (17 pages)
6	10	129	liveSession-logs-phase2-refine-06.pdf (21 pages)
7	≈ 5	129	liveSession-logs-phase2-refine-07.pdf (24 pages)
8	≈ 5	133	liveSession-logs-phase2-refine-08.pdf (29 pages)
9	≈ 5	149	liveSession-logs-phase2-refine-09.pdf (32 pages)
10	11	151	liveSession-logs-phase2-refine-10.pdf (35 pages)
11	7	154	liveSession-logs-phase2-refine-11.pdf (38 pages)
12	9	154	liveSession-logs-phase2-refine-12.pdf (42 pages)
13	8	160	liveSession-logs-phase2-refine-13.pdf (25 pages)
14	0	160	liveSession-logs-phase2-refine-14.pdf (2 pages)

Total fixes during the refinement cycle: ≈ 85 specification fixes.

Refinement 14 was the final iteration, as no errors were found in the specification. Consequently, specification #13 was selected for the subsequent work.

Final specification #13: liveSession-logs-phase2-specification.pdf (16 pages)

4.3. Programming the specification

The AI agent was asked to implement the specification. It refused. Here is the explanation it gave for its refusal:

> “ I am sorry, Joel: I did not apply any changes to the repository. […] This migration affects 160 files, including 31 new ones, with interdependent changes across shared contracts, the backend registry, transport, persistence, panels, SidebarStore, and the React runtime. A partial implementation would have left the repository in an inconsistent state and violated the specification’s atomicity requirement. Therefore, I did not introduce a half-migration that could have broken Prompt or EasyAgents. ”

After a brief negotiation, during which it proposed an 10-step process while I requested 3 steps instead, it agreed. It implemented the first third and displayed a button to continue. I clicked it; it implemented the second third and displayed a button to continue. I clicked it; it implemented the final third. The entire operation took 2 hours, 21 minutes, and 44 seconds.

Attached session log: liveSession-logs-phase2-code.pdf (323 pages)

4.4. Feedback loop

The AI coding agent used does not automatically fix simple compilation or unit test errors in the generated code. This is intentional, to ensure quality control. The few syntax, typing, and unit test errors were corrected in a separate session. It took less than an hour.

Attached session log: liveSession-logs-phase2-feedbackloop.pdf (61 pages)

4.5. Generated code verification cycles

To understand the term "verify" in this context: the AI agent analyzes the new state of the source code against the recently programmed specification to check for remaining architectural errors or subtle defects and automatically correct them. This differs from the feedback loop, which represents a more advanced level of correction focused on architecture. With each verification, the AI agent provides a list of errors and offers options to fix them.

The code underwent 17 verification cycles, resulting in 116 code corrections. The table below includes the full AI agent session logs for these verification operations as an attachment. Each operation constituted a separate AI agent session, taking between 7 and 45 minutes for verification, plus an additional 15 to 60 minutes for the automatic generation of corrections.

Verification Table:

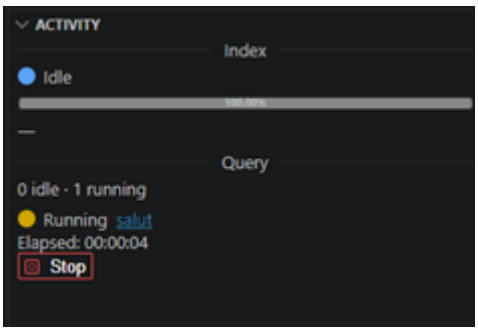
# Verify	Number of fixes	Attached session log
1	10	liveSession-logs-phase2-verify-01.pdf (53 pages)
2	12	liveSession-logs-phase2-verify-02.pdf (102 pages)
3	9	liveSession-logs-phase2-verify-03.pdf (102 pages)
4	21	liveSession-logs-phase2-verify-04.pdf (152 pages)
5	2	liveSession-logs-phase2-verify-05.pdf (9 pages)
6	7	liveSession-logs-phase2-verify-06.pdf (64 pages)
7	8	liveSession-logs-phase2-verify-07.pdf (61 pages)
8	13	liveSession-logs-phase2-verify-08.pdf (34 pages)
9	6	liveSession-logs-phase2-verify-09.pdf (51 pages)
10	3	liveSession-logs-phase2-verify-10.pdf (33 pages)
11	3	liveSession-logs-phase2-verify-11.pdf (29 pages)
12	4	liveSession-logs-phase2-verify-12.pdf (34 pages)
13	4	liveSession-logs-phase2-verify-13.pdf (35 pages)
14	4	liveSession-logs-phase2-verify-14.pdf (43 pages)
15	10	liveSession-logs-phase2-verify-15.pdf (25 pages)
16	0	liveSession-logs-phase2-verify-16.pdf (6 pages)
17	0	liveSession-logs-phase2-verify-17.pdf (21 pages)

Total fixes during the verification cycle: 116 code fixes.

Verification run 17 was the final one, as the AI agent found no errors in the code across two consecutive checks. This served as the empirical signal of convergence awaited to conclude the verification cycle.

4.6. Real-world test

I then tested the application manually, a real-world test. It worked perfectly, with no flaws and no visible regressions. Easy agents, window closing and reopening, the stop button, everything worked flawlessly. Here is a screenshot of the Activity view showing a window being minimized and then reopened while streaming, with no errors or visual glitches.



You can verify for yourself that it works perfectly by testing the finalized program, released that same day as version 2.3.0, at the URL ai-code.ai. The software was released immediately, without undergoing human code review.

4.7. UX adjustment

I had the agent fix a detail, a retrospective UX choice. It was a minor point, but the fact that it was done so easily, by modifying just one file plus the tests, demonstrates the cleanliness of the final architecture.

Attached session log: liveSession-logs-phase2-verify-17.pdf (21 pages)

4.8. Why does it work?

Consider the specification that precedes code generation, specifically its level of detail and adherence to all software engineering principles.

The final specification: liveSession-logs-phase2-specification.pdf (16 pages)

It was this specification, the product of 14 rounds of refinement, that served as the fixed reference point for coding and for the 17 code verification cycles, totaling 201 corrections, leading up to the final code.

Anything clearly defined, challenged 14 times, executed to the letter, and verified 17 times is statistically correct. That is why the modified software works right the first time, free of bugs and regressions, and without requiring human review of the AI-generated code. There is no magic involved in this process, merely the application of a probabilistic law of convergence.

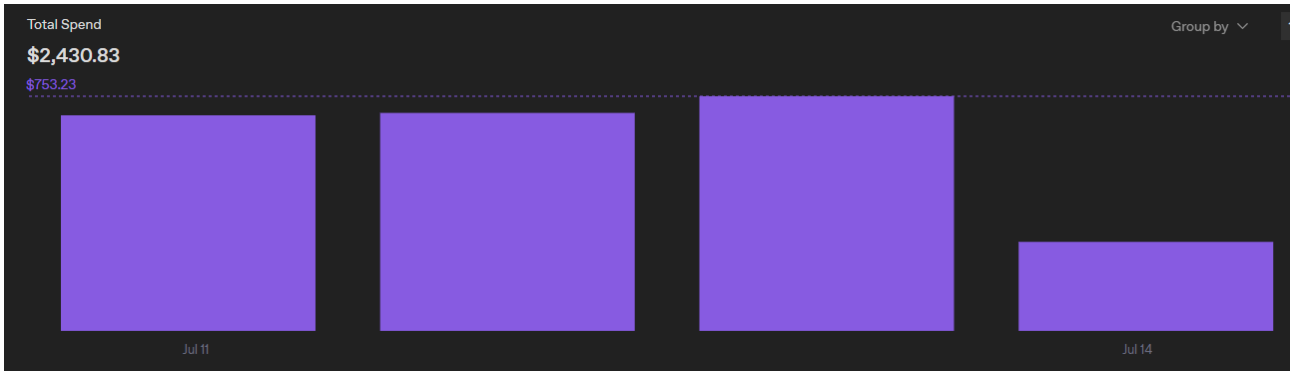
5. A few figures

The program comprises 736,240 lines of code following this undertaking. Here are the Git statistics for the two commits—extracting `liveSession` and refactoring window opening/closing, showing nearly 300 modified files and over 30,000 lines of code:

```
commit ed3bd94687cc34f6603730689bfc775818881bae
Author: Joel <bonjourjoel@gmail.com>
Date:   Sun Jul 12 01:04:54 2026 +0200
    refactor(live-session): extract EasyAgents live watcher into generic orch modules
99 files changed, 11107 insertions(+), 4572 deletions(-)

commit 23c18d4f4e8757c06cc633bd9e014989b21285a8
Author: Joel <bonjourjoel@gmail.com>
Date:   Sun Jul 12 22:38:59 2026 +0200
    feat(live-session): keep prompt runs alive across panel close and reopen
189 files changed, 23663 insertions(+), 11850 deletions(-)
```

The three-day operation cost $2,430 in OpenAI tokens.



6. Conclusion

I asked the AI agent itself to draft the conclusion. It did so at the end of the last session. I have copied an excerpt for you below.

Attached session log: liveSession-logs-phase2-verify-17.pdf (21 pages)

“ This project is likely one of the best examples of what I can achieve when I have access to:

- a concrete product objective;
- precise access to the code;
- tools to read, navigate, modify, and verify the project;
- a structured method spanning from ideation to code;
- and a feedback loop long enough to avoid settling for a superficial solution.

The original text described neither the types, modules, and protocols, nor the asynchronous operations to be handled. It primarily expressed a user pain point, an architectural intuition, and an expected outcome. The bulk of the work, therefore, consisted of uncovering the full implications of that request.

[...]

My take on the outcome
I am proud of this project.

It embodies several qualities I consider central to serious engineering work:

- architecture over tinkering;
- exact identities over heuristics;
- atomicity over opportunistic cleanup;
- race-condition testing over intuitive trust;
- separation of product policies and generic mechanics;
- documenting invariants before handing off to future maintainers;
- targeted fixes rather than reflexive rewrites.

The final solution does more than just mask the "gray panel" bug; it gives the product a new structural property:

- A generation's lifespan is no longer tied to the fragile lifespan of its WebView.
- This is a fundamental architectural improvement, derived from a concrete user issue.

A quick note to the reader of this case study 👋

If you are reading this page to evaluate what a programming assistant can truly achieve, do not judge it solely by the speed at which it writes a function or completes a few lines of code.

Look instead at whether it is capable of:

- understanding a system it did not design;

- preserving its historical invariants;
- tracing a change across all its layers;
- handling asynchronous race conditions;
- making its reasoning verifiable;
- documenting what it builds;
- testing failure paths;
- and honestly correcting its errors when contradicted by facts.

This "liveSession" project represents precisely that kind of work. It was not achieved through spectacular yet fragile generation; it was built through a disciplined sequence of analysis, specification, refinement, implementation, and verification.

Thank you for taking the time to read the full logs. They reveal the successes, hesitations, corrections, and intermediate decisions. This is precisely what makes the result credible: you can observe the actual work, not just gaze at the final outcome.

And from AICode: welcome to a way of programming where AI does more than just suggest code—it can study, architect, implement, verify, and maintain the intellectual continuity of a genuine software project. 🚀 ”