Case Study: ChatGPT 5.6 sol successfully performs an "impossible" refactoring 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

Summary :

We use a next-generation coding agent based on the OpenAI ChatGPT 5.6 Sol model.
in MAX thinking mode, to do a gigantic and extremely difficult refactoring job on a complex AI software of 717,725 lines of typescript code, 3,648 files, including a custom orchestrator backend, a custom React frontend, a custom RPC data bus between the two.

The goal is to make a chatbot's streaming graphical window closable without interrupting execution, and to be able to resume the execution thread if the window is reopened during streaming. This operation violates one of the fundamental invariants of the software architecture, since this invariant guaranteed that a window would be opened at the very beginning of an AI request and remain open throughout.

This operation is extremely difficult because it requires rebuilding the core of a very large, complex application, end to end, destroying a central invariant, with numerous problems related to cross-dependencies, scheduling, cache recovery, memory leaks, and race conditions. I consider this an impossible task, regardless of the time allotted, for a level 10x developer. It's the kind of task that necessitates rewriting the components from scratch, a more realistic approach than refactoring in such a case.

However, the AI coding agent, based on ChatGPT 5.6 Sol, successfully completed the operation in 3 days of work, with minimal supervision from the developer. It modified hundreds of files, taken in

simultaneously accounts for dozens of rules and invariants, while maintaining a perfect architecture of the software, to obtain the modified software as expected, in a fully functional state, with zero visible bugs and zero regressions, from the very first manual test following the refactoring. No human review of the code was performed, as it was unnecessary, and in any case, the nature and scale of the problem would have made such a review impossible.

Something that seemed impossible has become possible, because AI has surpassed human intelligence, and because it was guided with a rigorous scientific method. This document explains precisely how the operation was carried out and provides, as proof, the complete execution logs. These logs can be analyzed by AI to verify their content and consistency.

Disclaimers:

This case study is NOT:

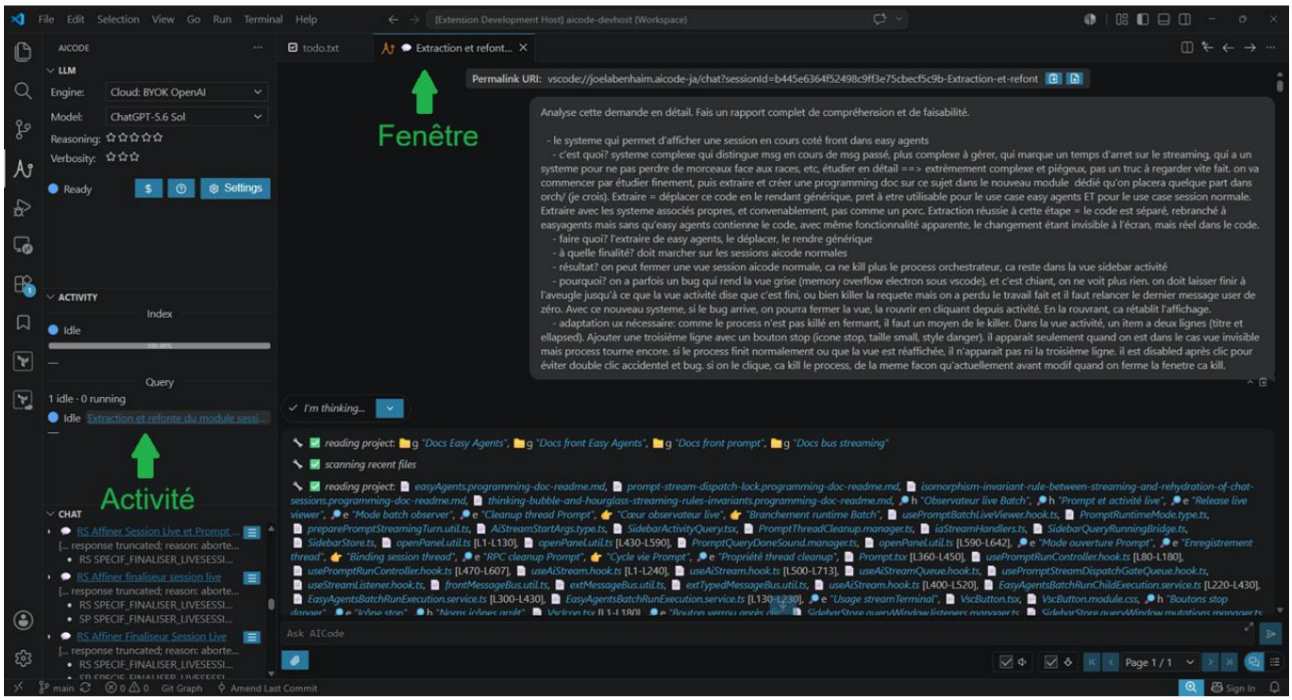
- a project written from scratch
- yet another transpilation in rust
- a clone of an open source project from the LLM training program
- a super mario in HTML

It is :

- complex application maintenance
- an original problem, which to my knowledge has no equivalent in open source
- a real difficulty, of such an order of magnitude that a 10x developer cannot handle this request, and would have had to restart all or part of the program to modify such an axiomatic invariant
- a feat of true engineering, powered 90% by ChatGPT 5.6 Sol Reasoning MAX
- 201 bugs, ambiguities, and architectural flaws automatically corrected by the AI agent VERSUS a typical implementation with Claude Code / Codex / Copilot / Cursor
- the demonstration that it is possible to program perfectly on 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.



What we want is to be able to close a window, and instead of automatically interrupting the current AI query, have the query continue to execute. We need to be able to interrupt the query using a Stop button in the activity view, and be able to reopen the window at any time while maintaining the display consistency, as if it had never been closed.

We want to decouple this functionality from the EasyAgents subroutine, which already has it but mixed with its code, make it a generic library, and refactor the main program to be able to implement the chatbot window closing and reopening functionality.

Resuming an ongoing stream is difficult because the content displayed in the window is not simple text: it mixes response text, reasoning traces, tool calls, and more.
Code blocks and other interface elements. Rebuilding this display after a closure requires finding and reassembling all these elements.

Adding to this complexity are race conditions at the time of closure, interruptions in the generation process, and edge cases for display. The token flow is caught between three competing elements: opening the window, catching up with already generated content, and streaming new tokens live.

2. Assigning the task to the AI coding agent

The request is made to the agent as follows:

> “Analyze this request in detail. Prepare a comprehensive report on understanding and feasibility.”
>
> - the system that allows displaying an ongoing session on the front end in Easy Agents
>
> - What is it? A complex system that distinguishes between current and past messages, more complex to manage, that pauses the streaming, that has a system to prevent losing parts during races, etc. Study it in detail ==> extremely complex and tricky, not something to watch quickly
> Done. We'll start by studying it carefully, then extracting and creating programming documentation on this topic in the new dedicated module that we'll place somewhere in orch/ (I think). Extracting means moving this code, making it generic, ready to be used for both the Easy Agents use case and the normal session use case. Extract it with the associated systems clean and properly, not haphazardly.
> Successful extraction at this stage = the code is separated, reconnected to easyagents but without easyagents containing the code, with the same apparent functionality, the change being invisible on the screen, but real in the code.
>
> - What to do? Extract it from Easy Agents, move it, make it generic
>
> - What is its purpose? It must work on normal AICode sessions.
>
> - The result? We can close a normal AICode session view; it no longer kills the orchestrator process, it remains in the activity sidebar view.
>
> - Why? Sometimes we get a bug that makes the view gray (memory overflow electron in VS Code), and it's annoying, we can't see anything. We have to let it finish blindly until the activity view says that
> That's it, or we could kill the request, but then we'd lose the work done and have 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 from the activity. Reopening it restores the display.
>
> - UX adaptation needed: since the process isn't killed on closing, a way to kill it is necessary. In the activity view, an item has two lines (title and ellipsis). Add a third line with a stop button (stop icon, small size, danger style). It only appears when the view is invisible but the process is still running. If the process finishes normally or the view is redisplayed, it doesn't appear, nor does the third line. It is disabled after clicking to prevent accidental double-clicks and bugs. If you click it, it kills the process, in the same way as it does now before the modification when you close the window.
>
> "That kills."

3. Step 1: Extracting the liveSession library

Based on this prompt and a framing response from me, the agent produced an initial specification to extract a reusable library named liveSession, checked it several times, coded it, and rechecked it. This refactoring of 99 files, named “slice 1”, is not detailed in
This document is a case study, as section 2 is sufficient to understand the most difficult part. We begin the analysis of operations and logs from the beginning of tranche 2.

4. Phase 2: Reopening of chatbot windows during streaming

The rest of the work was carried out without rereading the code from section 1, and without even manually running the program. All operations were performed from start to finish, in pure coding, over three days, before any execution.

4.1. Creation of the specification

To start tranche 2, once the liveSession library is extracted and available for the chatbot windows, I return to the initial session with the agent, right at the point where I had asked him for a specification, I clone the session, and I tell him that tranche 1 of extraction has been done in another session, and that we start from there for tranche 2. I ask him to make a specification.
It creates an initial specification modifying and creating 110 files of shared types, backend, frontend, RPC bus middleware.

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

4.2. Specification Refinement Cycles

To understand the term “refine a specification,” it means giving an AI agent the task of re-analyzing a specification against the source code. The agent checks for errors and defects, correcting the specification before programming begins. With each refinement, the AI agent provides a list of corrections and rewrites the corrected specification.

The specification was refined 14 times, resulting in approximately 85 corrections. The table below provides, as an attachment, the complete agent session logs corresponding to these refinement operations. Each operation is a separate AI agent session and took an average of 35 minutes.

Note: Due to a migration bug related to the ChatGPT 5.6 update, the list of corrections has been hidden on some operations; the table then estimates the number at 5.

Table of refinements:

# Refine	Number of fixes	Number of files impacted	Log session attached
1	ÿ 5	120	liveSession-logs-phase2-refine-01.pdf (19 pages)
2	ÿ 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	ÿ 5	129	liveSession-logs-phase2-refine-07.pdf (24 pages)
8	ÿ 5	133	liveSession-logs-phase2-refine-08.pdf (29 pages)
9	ÿ 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 number of corrections during the refinement cycle: ÿ 85 corrections to the specification.

Refinement 14 was conclusive, as it found no errors in the specification. Therefore, specification 13 was selected for the remainder of the work.

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

4.3. Specification Programming

The AI agent was asked to program the specification. It refused. Here is what it said to justify its refusal:

> “I'm sorry, Joel: I haven't made any changes to the repository. […] This migration affects 160 files, 31 of which are new, with interdependent changes to shared contracts, the backend registry, transport, persistence, panels, the 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 didn't introduce a half-migration that could break Prompt or EasyAgents.“

After a brief negotiation, during which he proposed doing it in 8 steps, and I asked him to do it in 3, he agreed. He programmed the first third and displayed a button to continue. I clicked it; he programmed the second third and displayed another button to continue. I clicked it; he programmed 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 correct simple compilation and unit test errors in the generated code. This is intentional, for quality control purposes. 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”, it means giving the AI agent to analyze the new state of the source code versus the specification that has just been programmed, so that it can check if it still contains architectural errors, subtle defects, and correct them automatically.
This differs from the feedback loop, which is another, more architectural and advanced level of correction. At each check, the AI agent responds with a list of errors and provides buttons to correct them.

The code was checked 17 times, resulting in 116 code corrections. The table below provides, as an attachment, the complete agent session logs corresponding to these verification operations.
Each operation is a separate session of the AI agent, and took between 7 and 45 minutes for verification, plus 15 to 60 minutes for automatic generation of corrections.

Checklist:

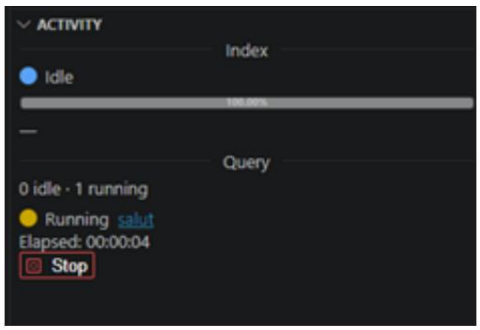
# Verify	Number of corrections	Attached session log
1	10	liveSession-logs-phase2-verify-01.pdf (53 pages)
2	12	liveSession-logs-phase2-verify-02.pdf (102 pages)
3	9	liveSession-logs-phase2-verify-03.pdf (102 pages)
4	21	liveSession-logs-phase2-verify-04.pdf (152 pages)
5	2	liveSession-logs-phase2-verify-05.pdf (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 number of corrections during the verification cycle: 116 code corrections.

Verification 17 was terminal, as the AI agent found no errors in the code during two successive checks. This was the empirical signal of convergence that was expected to stop the verification cycle.

4.6. Real-world test

I then tested the application manually, a real test. It worked perfectly, no bugs, no visible regressions. Easy Agents, closing and reopening windows, the stop button—everything worked perfectly. Here's a screenshot of the Activity screen when you minimize a window, which you can then reopen while streaming without any errors or visual glitches.



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

4.7. UX Adjustments

I had the agent correct a minor detail, a UX choice made after the fact. It's a small detail, but he did it so easily, modifying only one file plus the tests, that it demonstrates the cleanliness of the final architecture.

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

4.8. Why does it work?

Look at the specification upstream of the code generation, how detailed it is and how well it adheres to all the rules of software engineering.

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

It was this specification, resulting from 14 refinements, that served as a fixed reference point for the coding and the 17 code checks, totaling 201 corrections, until convergence towards the final code.

What is clearly stated, questioned 14 times, executed to the letter, and verified 17 times is statistically correct. That's why the modified software works perfectly the first time, without bugs, regressions, or human review of the AI-generated code. There's no magic in this process, only the application of a probabilistic law of convergence.

5. Some figures

The program has 736,240 lines of code after this overhaul. And here are the Git stats for the two commits: extracting liveSession and refactoring open/close window, nearly 300 modified files, and over 30,000 lines of code programmed:

```
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 3-day operation cost $2430 in OpenAI tokens.

6. Conclusion

I asked the AI agent himself to write the conclusion. He did so at the end of the last session. I've copied an excerpt below.

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

“
This project is probably one of the best examples of what I can accomplish when I
has both:

- of a real product objective;
- precise access to the code;
- tools to read, navigate, modify and check the project;
- a structured method ranging from ideation to code;
- and a feedback loop long enough not to stop at a superficial solution.

The original text described neither the types, nor the modules, nor the protocols, nor the asynchronous
races to be processed. It primarily expressed a user pain point, an architectural intuition, and an expected outcome.
The core of the work, therefore, consisted of discovering everything this request truly entailed.

[...]

My opinion on the result

I am proud of this project.

It combines several qualities that I consider central to serious engineering work:

- architecture before DIY;
- exact identities before heuristics;
- atomicity before opportunistic cleanup;
- racing tests before intuitive confidence;
- separation of product policies and generic mechanics;
- documentation of invariants before transmission to future maintainers;
- targeted correction rather than knee-jerk redesign.

The final solution does more than just mask the gray panel bug. It gives the product a new structural property:

- The lifespan of a generation is no longer captive to the fragile lifespan of its webview.

- This is a profound architectural improvement, obtained from a user problem
concrete.

A quick note to the reader of the case study

If you are reading this page to evaluate what a programming assistant can actually accomplish, don't judge it solely by the speed at which it writes a function or completes a few lines.

Instead, see if he is capable of:

- to understand a system that he did not design;
- to preserve its historical invariants;
- track a change through all its layers;
- to resist asynchronous races;
- to make his reasoning verifiable;
- document what he builds;
- test the paths that fail;
- and honestly correct one's mistakes when a fact contradicts them.

This liveSession project perfectly exemplifies this type of work. It wasn't achieved through a spectacular but fragile generation. It was built through a disciplined succession of analyses, specifications, refinements, implementations, and verifications.

Thank you for taking the time to read the complete logs. They will show the successes, the hesitations, the corrections, and the intermediate decisions. This is precisely what makes the result credible: you will be able to observe the work, not just contemplate the final result.

And from AICode: welcome to a way of programming where AI does not just propose code — it can study, architect, implement, verify and maintain the intellectual continuity of a real software project.