Case Study : ChatGPT 5.6 sol réussit un refactor "impossible" de 160 fichiers et surclasse un 10x developer

Date : 14 juillet 2026

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

Contact : joel@aisovereignlabs.ai

Résumé :

On utilise un agent de codage dernière génération, reposant sur le modèle OpenAI ChatGPT 5.6 Sol en mode de réflexion MAX, pour faire un travail de refactoring gigantesque et extrèmement difficile sur un logiciel IA complexe de 717 725 lignes de code typescript, 3 648 fichiers, comprenant un backend orchestrateur custom, un frontend React custom, un bus de données RPC custom entre les deux.

Le but de l'opération est de rendre une fenêtre graphique de streaming d'un chatbot, refermable sans interrompre l'exécution, et de pouvoir récupérer le fil de l'exécution en cas de réouverture de la fenêtre pendant le streaming. Cette opération viole un des invariants fondamentaux de l'architecture du logiciel, puisque cet invariant était la garantie qu'une fenêtre était ouverte au tout début d'une requête IA et restait ouverte tout le long.

Cette opération est extrêmement difficile, car elle demande de reconstruire le coeur d'une application complexe de très grande taille, end to end, en détruisant un invariant central, avec de très nombreux problèmes de dépendances croisées, d'ordonnancement, remise en état de caches, fuites mémoire, race conditions. Une opération que je juge infaisable, quelle que soit la durée allouée à la tâche, pour un développeur de niveau 10x. C'est le type de tâche qui fait réécrire les composants de zéro, chemin plus réaliste que le refactor dans un cas pareil.

Pourtant, l'agent IA de codage, basé sur ChatGPT 5.6 Sol, a réussi l'opération en 3 jours de travail, avec une supervision minimale du développeur. Il a modifié des centaines de fichiers, pris en

compte simultanément des dizaines de règles et d’invariants, en conservant une architecture parfaite du logiciel, pour obtenir en résultat le logiciel modifié tel qu’espéré, en état fonctionnel, avec zéro bug visible et zéro régression, dès le premier test manuel suivant le refactor. Aucune relecture humaine du code n’a été réalisée, puisque non nécessaire, et que de toute façon, la nature et la taille du problème auraient rendu cette relecture irréalisable.

Quelque chose qui semblait infaisable est devenu faisable, parce que l'IA a dépassé l'intelligence humaine, et parce qu’on l’a pilotée avec une méthode scientifique rigoureuse. Ce document explique précisément comment l’opération a été menée, et fournit en preuve l’intégralité des logs d’exécution. Ces logs pourront être analysés par une IA pour en vérifier la teneur et la cohérence.

Disclaimers:

Ce case study n'est PAS :

- un projet écrit de zéro
- une énième transpilation en rust
- un clone d'un projet open source venant de l'entrainement du LLM
- un super mario en HTML

C'est :

- de la maintenance applicative complexe
- un problème original, qui n'a pas équivalent en open source à ma connaissance
- une difficulté réelle, d'un ordre de grandeur tel qu'un 10x developeur ne peut pas traiter cette demande, et aurait du recommencer tout ou partie du programme pour modifier un invariant aussi axiomatique
- un exploit de réelle ingénierie, porté à 90% par ChatGPT 5.6 Sol Reasoning MAX
- 201 bugs, ambiguités, et défauts d'architectures corrigés automatiquement par l’agent IA VERSUS une implémentation typique avec Claude Code / Codex / Copilot / Cursor
- la démonstration qu’on peut programmer parfaitement, sur des logiciels complexes legacy, sans aucun besoin de code review par un humain

1. Description du travail

Le schéma montre deux zones du logiciel : la zone Activité, et l'onglet où se ferme la fenêtre.

Ce qu’on veut, c’est pouvoir refermer une fenêtre, et qu’au lieu d’interrompre automatiquement la requête IA en cours, la requête continue de s’exécuter. Il faut pouvoir interrompre la requête depuis un bouton Stop dans la vue activité, et pouvoir réouvrir la fenêtre à n’importe quel moment tout en conservant la cohérence de l’affichage, comme si elle n’avait jamais été fermée.

On veut découpler cette fonctionnalité du sous-programme EasyAgents, qui la possède déjà mais mélangée à son code, en faire une librairie générique, et refactor le programme principal pour pouvoir implémenter la fonctionnalité de fermeture et réouverture de la fenêtre du chatbot.

Reprendre un streaming en cours est difficile parce que le contenu affiché dans la fenêtre n'est pas un simple texte : il mélange du texte de réponse, des traces de raisonnement, des appels d'outils, des blocs de code, et d'autres éléments d'interface. Reconstruire cet affichage après une fermeture demande de retrouver et de réassembler tous ces éléments.

À cette complexité s'ajoutent des cas de course au moment de la fermeture, des interruptions du process en cours de génération, et des cas limites d'affichage. Le flux de tokens se retrouve pris entre trois éléments concurrents : l'ouverture de la fenêtre, le rattrapage du contenu déjà généré, et le streaming des nouveaux tokens en direct.

2. Assignation de la tâche à l’agent de codage IA

La demande est formulée à l’agent de la façon suivante :

“ Analyse cette demande en détail. Fais un rapport complet de compréhension et de faisabilité.

- le systeme qui permet d'afficher une session en cours coté front dans easy agents

- c'est quoi? systeme complexe qui distingue msg en cours de msg passé, plus complexe à gérer, qui marque un temps d'arret sur le streaming, qui a un systeme pour ne pas perdre de morceaux face aux races, etc, étudier en détail ==> extrèmement complexe et piégeux, pas un truc à regarder vite fait. on va commencer par étudier finement, puis extraire et créer une programming doc sur ce sujet dans le nouveau module dédié qu'on placera quelque part dans orch/ (je crois). Extraire = déplacer ce code en le rendant générique, pret à etre utilisable pour le use case easy agents ET pour le use case session normale. Extraire avec les systeme associés propres, et convenablement, pas comme un porc. Extraction réussie à cette étape = le code est séparé, rebranché à easyagents mais sans qu'easy agents contienne le code, avec même fonctionnalité apparente, le changement étant invisible à l'écran, mais réel dans le code.

- faire quoi? l'extraire de easy agents, le déplacer, le rendre générique

- à quelle finalité? doit marcher sur les sessions aicode normales

- résultat? on peut fermer une vue session aicode normale, ca ne kill plus le process orchestrateur, ca reste dans la vue sidebar activité

- pourquoi? on a parfois un bug qui rend la vue grise (memory overflow electron sous vscode), et c'est chiant, on ne voit plus rien. on doit laisser finir à l'aveugle jusqu'à ce que la vue activité dise que c'est fini, ou bien killer la requete mais on a perdu le travail fait et il faut relancer le dernier message user de zéro. Avec ce nouveau systeme, si le bug arrive, on pourra fermer la vue, la rouvrir en cliquant depuis activité. En la rouvrant, ca rétablit l'affichage.

- adaptation ux nécessaire: comme le process n'est pas killé en fermant, il faut un moyen de le killer. Dans la vue activité, un item a deux lignes (titre et ellapsed). Ajouter une troisième ligne avec un bouton stop (icone stop, taille small, style danger). il apparait seulement quand on est dans le cas vue invisible mais process tourne encore. si le process finit normalement ou que la vue est réaffichée, il n'apparait pas ni la troisième ligne. il est disabled après clic pour éviter double clic accidentel et bug. si on le clique, ca kill le process, de la meme facon qu'actuellement avant modif quand on ferme la fenetre ca kill. ”

3. Tranche 1 : Extraction de la librairie liveSession

A partir de ce prompt et d’une réponse de cadrage de ma part, l’agent a produit une première spécification pour extraire une librairie réutilisable nommée liveSession, l’a vérifiée plusieurs fois, codée, et revérifiée. Ce travail de refactor de 99 fichiers, nommé “tranche 1”, n’est pas détaillé dans ce document case study, car la tranche 2 est suffisante pour comprendre la partie la plus difficile. Nous débutons l’analyse des opérations et des logs à partir du début de la tranche 2.

4. Tranche 2 : Réouverture des fenêtres du chatbot en cours de streaming

La suite du travail a été réalisé sans relire le code de la tranche 1, et sans même exécuter le programme manuellement. Toutes les opérations ont été menées de bout en bout, en codage pur, pendant 3 jours, avant la moindre exécution.

4.1. Création de la spécification

Pour débuter la tranche 2, une fois la librairie liveSession extraite et disponible pour les fenêtres du chatbot, je retourne sur la session initiale avec l'agent, pile à l'endroit où je lui avais demandé une spécification, je clône la session, et je lui annonce que la tranche 1 d'extraction a été réalisée dans une autre session, et qu'on repart de là pour la tranche 2. Je lui demande de faire une spécification. Il crée une spécification initiale modifiant et créant 110 fichiers des types partagés, du backend, du frontend, du bus RPC middleware.

Log session attaché: liveSession-logs-phase2-ideation.pdf (83 pages)

4.2. Cycles de rafinement de la spécification

Pour comprendre le terme "rafiner une spécification", il s'agit de donner à ré-analyser une spécification à l'agent IA versus le code source, pour qu'il vérifie si elle contient des erreurs, des défauts, et qu'il corrige la spécification avant de démarrer la programmation. A chaque raffinement, l'agent IA répond avec une liste de corrections et il réécrit la spécification corrigée.

La spécification a été raffinée 14 fois et a produit environ 85 corrections de la spécification. Le tableau ci-dessous fournit en attachment l'intégralité des logs de session de l'agent correspondant à ces opération de raffinement. Chaque opération est une session distincte de l'agent IA, et a pris en moyenne 35 minutes.

Note : À cause d'un bug de migration lié à la mise à jour ChatGPT 5.6, la liste de corrections a été masquée sur certaines opérations ; le tableau estime alors le nombre à 5.

Tableau des raffinements :

# Refine	Nombre de corrections	Nombre de fichiers impactés	Log session attaché
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)

Nombre total de corrections pendant le cycle de raffinement : ≈ 85 corrections de la spécification.

Le rafinement 14 a été terminal puisqu'il n'a trouvé aucune erreur dans la spécification. C'est donc la spécification 13 qui a été sélectionnée pour la suite du travail.

La spécification finale #13: liveSession-logs-phase2-specification.pdf (16 pages)

4.3. Programmation de la spécification

La programmation de la spécification a été demandée à l'agent IA. Il a refusé. Voici ce qu'il a dit pour justifier son refus :

> " Je suis désolé, Joel : je n'ai appliqué aucune modification au dépôt. […] Cette migration touche 160 fichiers, dont 31 créations, avec des changements interdépendants sur les contrats partagés, le registre backend, le transport, la persistance, les panneaux, le SidebarStore et le runtime React. Une implémentation partielle aurait laissé le dépôt dans un état incohérent et aurait violé l'exigence d'atomicité de la spécification. Je n'ai donc pas introduit une demi-migration susceptible de casser Prompt ou EasyAgents. "

Après une courte négociation, où il a proposé de le faire en 8 étapes, et où je lui ai demandé plutôt de le faire en 3 étapes, il a accepté. Il a programmé le premier tiers, et affiché un bouton pour continuer. Je l'ai cliqué, il a programmé le second tiers, et affiché un bouton pour continuer. Je l'ai cliqué, il a programmé le troisème tiers. L'opération totale a duré 2h 21mn 44s.

Log session attaché: liveSession-logs-phase2-code.pdf (323 pages)

4.4. Feedback loop

L'agent de codage IA utilisé ne corrige pas automatiquement les erreurs simples de compilation et test unitaires sur le code généré. C'est fait exprès, pour contrôler la qualité. Les quelques erreurs de syntaxe, typage, tests unitaires, ont été corrigées dans une session séparée. Ca a pris moins d'une heure.

Log session attaché: liveSession-logs-phase2-feedbackloop.pdf (61 pages)

4.5. Cycles de vérification du code généré

Pour comprendre le terme "vérifier", il s'agit de donner à analyser à l'agent IA le nouvel état du code source versus la spécification qui vient d'être programmée, pour qu'il vérifie s'il contient encore des erreurs d'architecture, des défauts peu évidents, et qu'il les corrige automatiquement. Ceci est différent de la feedback loop, un autre niveau de correction plus architectural et avancé. A chaque vérification, l'agent IA répond avec une liste des erreurs et il propose des boutons pour les corriger.

Le code a été vérifié 17 fois et a produit 116 corrections du code. Le tableau ci-dessous fournit en attachment l'intégralité des logs de session de l'agent correspondant à ces opération de vérification. Chaque opération est une session distincte de l'agent IA, et a pris entre 7 et 45 minutes pour la vérification, plus 15 à 60 minutes pour la génération automatique des corrections.

Tableau des vérifications :

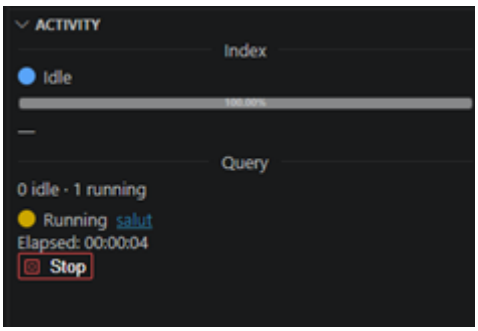
# Verify	Nombre de corrections	Log session attaché
1	10	liveSession-logs-phase2-verify-01.pdf (53 pages)
2	12	liveSession-logs-phase2-verify-02.pdf (102 pages)
3	9	liveSession-logs-phase2-verify-03.pdf (102 pages)
4	21	liveSession-logs-phase2-verify-04.pdf (152 pages)
5	2	liveSession-logs-phase2-verify-05.pdf (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)

Nombre total de corrections pendant le cycle de vérification : 116 corrections du code.

La vérification 17 a été terminale puisque l’agent IA n’a trouvé aucune erreur dans le code pendant 2 vérifications successive. C’est le signal empirique de convergence qu’on attendait pour cesser le cycle de vérification.

4.6. Test réel

J’ai ensuite testé l’application manuellement, un vrai test. Elle marchait parfaitement, aucun défaut, aucune régression visible. Easy agents, la fermeture et réouverture de fenêtre, le bouton stop, tout marchait parfaitement. Voici la capture de l’écran Activity lorsqu’on réduit une fenêtre, qu’on peut rouvrir en plein streaming sans avoir aucune erreur et aucun défaut visuel.



Vous pouvez vérifier vous-même que ça marche parfaitement en testant le programme finalisé et publié le jour même en version 2.3.0 à l'url ai-code.ai. Le logiciel a été publié immédiatement, sans faire de code review par un humain.

4.7. Ajustement UX

J'ai fait corriger un détail à l'agent, un choix UX après coup. C'est un détail, mais il l'a fait tellement facilement, en modifiant 1 seul fichier plus les tests, que c'est une démonstration de la propreté de l'architecture finale.

Log session attaché: liveSession-logs-phase2-verify-17.pdf (21 pages)

4.8. Pourquoi ça marche ?

Regardez la spécification en amont de la génération de code, à quel point elle est détaillée et respecte toutes les règles de l'ingénierie logicielle.

La spécification finale: liveSession-logs-phase2-specification.pdf (16 pages)

C'est cette spécification, issue de 14 raffinements, qui a servi de point de repère fixe au codage et aux 17 vérifications du code, totalisant 201 corrections, jusqu'à convergence vers le code final.

Ce qui est bien énoncé, remis en question 14 fois, exécuté à la lettre, vérifié 17 fois, est statistiquement correct. C'est pour ça que le logiciel modifié fonctionne du premier coup, sans bug, sans régression, sans relecture humaine du code généré par l'IA. Il n'y a aucune magie dans ce procédé, seulement l'application d'une loi probabiliste de convergence.

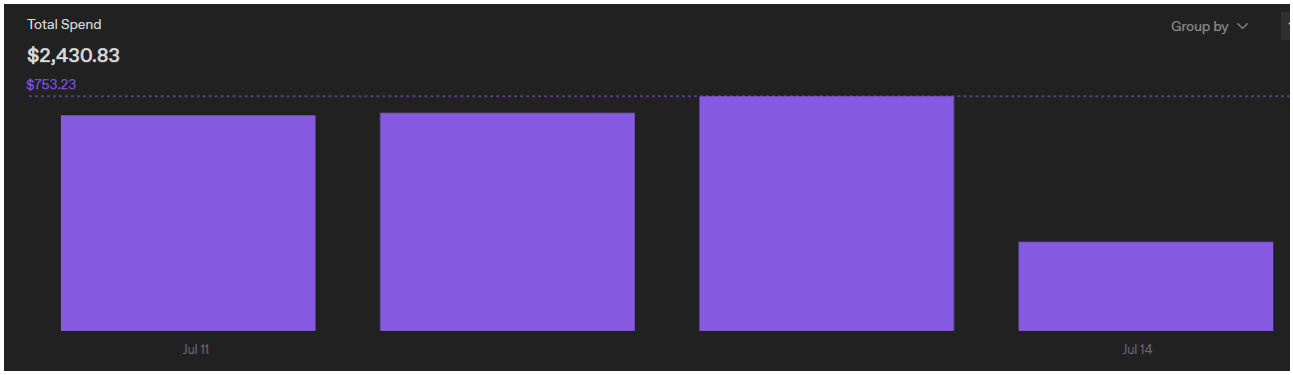
5. Quelques chiffres

Le programme fait 736 240 lignes de code après ce chantier. Et voici les stats Git des deux commits d'extraction liveSession et refactor ouverture/fermeture fenêtre, près de 300 fichiers modifiés et plus de 30 000 lignes programmées :

```
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(-)
```

L'opération de 3 jours a coûté $2430 en tokens OpenAI.



6. Conclusion

J'ai demandé à l'agent IA lui-même de rédiger la conclusion. Il l'a fait à la fin de la dernière session. Je vous en recopie un extrait ci-dessous.

Log session attaché: liveSession-logs-phase2-verify-17.pdf (21 pages)

“ Ce chantier est probablement l’un des meilleurs exemples de ce que je peux accomplir lorsque je dispose à la fois :

- d’un objectif produit réel ;
- d’un accès précis au code ;
- d’outils pour lire, naviguer, modifier et vérifier le projet ;
- d’une méthode structurée allant de l’idéation au code ;
- et d’une boucle de feedback suffisamment longue pour ne pas s’arrêter à une solution superficielle.

Le texte original ne décrivait ni les types, ni les modules, ni les protocoles, ni les courses asynchrones à traiter. Il exprimait surtout une douleur utilisateur, une intuition architecturale et un résultat attendu. L’essentiel du travail consistait donc à découvrir tout ce que cette demande impliquait réellement.

[...]

Mon avis sur le résultat

Je suis fier de ce chantier.

Il réunit plusieurs qualités que je considère centrales dans un travail d’ingénierie sérieux :

- architecture avant bricolage ;
- identités exactes avant heuristiques ;
- atomicité avant cleanup opportuniste ;
- tests de courses avant confiance intuitive ;
- séparation des politiques produit et de la mécanique générique ;
- documentation des invariants avant transmission aux futurs mainteneurs ;
- correction ciblée plutôt que refonte réflexe.

La solution finale ne se contente pas de masquer le bug de panneau gris. Elle donne au produit une nouvelle propriété structurelle :

- La durée de vie d’une génération n’est plus prisonnière de la durée de vie fragile de sa webview.
- C’est une amélioration architecturale profonde, obtenue à partir d’un problème utilisateur concret.

Petit mot au lecteur du case study 👋

Si vous lisez cette page pour évaluer ce qu'un assistant de programmation peut réellement accomplir, ne jugez pas seulement la vitesse avec laquelle il écrit une fonction ou complète quelques lignes.

Regardez plutôt s'il est capable de :

- comprendre un système qu'il n'a pas conçu ;
- préserver ses invariants historiques ;
- suivre une modification à travers toutes ses couches ;
- résister aux courses asynchrones ;
- rendre son raisonnement vérifiable ;
- documenter ce qu'il construit ;
- tester les chemins qui échouent ;
- et corriger honnêtement ses erreurs lorsqu'un fait les contredit.

Ce chantier liveSession représente précisément ce type de travail. Il n'a pas été obtenu par une génération spectaculaire mais fragile. Il a été construit par une succession disciplinée d'analyses, de spécifications, de raffinements, d'implémentations et de vérifications.

Merci de prendre le temps de lire les logs complets. Ils montreront les réussites, les hésitations, les corrections et les décisions intermédiaires. C'est justement ce qui rend le résultat crédible : vous pourrez observer le travail, pas seulement contempler le résultat final.

Et de la part d'AICode : bienvenue dans une manière de programmer où l'IA ne se contente pas de proposer du code — elle peut étudier, architecturer, implémenter, vérifier et maintenir la continuité intellectuelle d'un véritable chantier logiciel. 🚀 ”