Un agent de codage n'est pas un chatbot

Traiter un agent de codage comme un agent de travail plutôt que comme un chatbot et ce que ça change concrètement.

2026-05-03 · Agent de codage · Workflow · Développeurs

Résumé

Un agent de codage donne de meilleurs résultats quand on cesse de l'utiliser comme un chatbot et qu'on le traite comme un agent de travail intégré au repository.

Le point clé n'est pas de trouver le prompt parfait, mais de construire un environnement lisible : contexte projet explicite, tâche bornée, validations exécutables, permissions adaptées au risque et review humaine sur les décisions importantes.

En pratique, cela change la façon de travailler : on documente les conventions dans le repo, on formule des briefs vérifiables, on choisit la bonne surface de travail selon le niveau de risque, et on demande à l'agent de produire des preuves de validation plutôt qu'un simple diff plausible.

Sommaire

Contexte de lecture

Audience

Cet article s'adresse aux développeurs, tech leads et architectes qui veulent dépasser l'usage "chatbot" d'un agent de codage et obtenir des résultats fiables dans un repository réel.

Ce que cet article couvre

Pourquoi le modèle "chatbot" atteint vite ses limites dans un repo réel, ce que j'entends par agent de travail, et quelques points d'entrée concrets pour expérimenter, quel que soit l'outil utilisé.

Ce que cet article ne couvre pas

Ce n'est pas une recette universelle, pas un comparatif d'agents, pas une promesse d'autonomie magique. C'est ma façon de faire, exposée pour que chacun puisse s'en inspirer et l'adapter à la sienne.

Une bascule mentale, plus qu'un changement d'outil

Quand on commence à utiliser un agent de codage sur des projets réels (pas des bouts de scripts, pas des notebooks d'exploration, mais du code qui finit en review et en prod) on met généralement un moment à comprendre pourquoi certaines sessions étaient excellentes et d'autres franchement moins concluantes.

Au début, on met cela sur le compte du modèle. Puis sur le compte de nos prompts. On essaye de les rendre plus précis, plus longs, plus structurés. Généralement cela aide, évidemment. Mais pas autant qu'on l'espère.

Un agent de codage a besoin d'être traité comme un coéquipier à qui l'on confie du travail, avec tout ce que ça implique en terme de contexte, de périmètre, de validation, de feedback, de review et de responsabilités.

OpenAI le formule à peu près ainsi dans ses best practices : Codex fonctionne mieux quand on le traite moins comme un assistant ponctuel et davantage comme un teammate que l'on configure et que l'on améliore dans le temps.[1] C'est aussi la conclusion à laquelle j'étais arrivé empiriquement, avant je l'avoue humblement d'avoir appliqué le fameux RTFM. Mais formulée comme cela, l'idée reste encore un peu abstraite. Ce qui m'intéresse ici, c'est ce que cette bascule change concrètement dans la façon de travailler avec un agent de codage.

Je précise le cadre : ce que je décris ici est ma méthode, dans mon contexte. Ce n'est pas une méthode universelle. Dans mon cas, j'utilise Codex parce que c'est l'outil qui me donne aujourd'hui les meilleurs résultats et le meilleur équilibre entre autonomie, contrôle et intégration au repository. Mais le propos n'est pas spécifique à Codex. La logique derrière, expliciter le contexte, borner la tâche, valider le résultat, relire ce qui compte, me semble largement transposable à Claude Code, Gemini, Cursor, Aider, Copilot Workspace ou d'autres agents de codage. Donc n'y voyez pas une incitation à utiliser Codex spécifiquement, mais plutôt une invitation à réfléchir à la façon dont vous utilisez votre agent de codage, et à ce que vous pourriez gagner à le traiter davantage comme un agent de travail intégré à votre repository.

Le piège du modèle "chatbot" et ce qui se passe vraiment

Le modèle chatbot est naturel. On a tous appris à utiliser un LLM en lui parlant : une demande, une réponse, une correction, puis une autre demande. Tant que la tâche est petite, isolée et facile à vérifier, cela fonctionne très bien. Sauf que ce que je viens de décrire est du vibe coding, pas du work coding.

Je continue d'ailleurs à travailler comme cela dans beaucoup de cas : explorer une API que je ne connais pas, comprendre un message d'erreur obscur, esquisser une architecture rapidement, comparer deux approches, reformuler un morceau de documentation, sortir une démo, un POC ou un prototype jetable. Pour ce type d'usage, le chatbot est efficace, et je ne vois aucune raison de s'en priver. Tout ce qui n'est pas appelé à devenir du code de production peut être traité comme ça.

Le piège commence quand on conserve ce mode de fonctionnement dans un repository réel.

Je demande à l'agent quelque chose comme :

```
Améliore ce composant.
```

Le diff arrive. Il est propre. Le code compile. À première vue, tout semble correct. Puis Je commence à relire plus attentivement.

Il a renommé une prop publique.

Il a extrait un helper dans un fichier où l'équipe ne met jamais ce type de helper.

Il a remplacé une lib utilitaire par une autre, parce qu'il a estimé que c'était plus propre.

On ne peut pas vraiment dire qu'il a halluciné. Il n'a pas inventé une API inexistante, ni produit du code absurde. Il a simplement interprété « améliore » comme l'aurait fait un développeur arrivé la veille sur le projet : avec de bonnes intentions, mais sans connaître les conventions tacites, les décisions passées, les zones sensibles et les petites règles non écrites qui se sont accumulées au fil des PR.

Le résultat est plausible, mais pas exploitable tel quel.

Pour le faire passer en review, soit j'accepte ces changements collatéraux (et j'accepte également que je devrais payer la dette plus tard) soit je reviens en arrière à la main. Dans le second cas, je finis systématiquement par perdre plus de temps que si j'avais écrit le code moi-même.

Ce n'est pas une question d'intelligence du modèle. C'est une question d'information disponible.

Mes conventions, mes « do not », l'histoire du module, les pièges connus, les raisons pour lesquelles tel fichier est structuré de telle façon : tout cela vit souvent dans la tête de l'équipe, dans des discussions Slack, dans des décisions de PR passées, dans une mémoire collective plus ou moins explicite. L'agent ne peut pas le deviner.

Tant que je continue à lui transmettre ce contexte au fil de la conversation, chaque session repart presque de zéro. Je porte tout l'implicite à la main, comme si je devais réexpliquer le projet à chaque nouvelle tâche.

Le vrai coût du modèle chatbot dans un repo réel, c'est celui-là : il transforme chaque session en exercice de transmission d'implicite. On croit gagner du temps sur l'écriture du code, mais on en perd à corriger les interprétations, à reprendre les écarts, à remettre les conventions dans la conversation.

Ce n'est pas le prompt, c'est le système

La transition la plus efficace pour moi, a été d'arrêter de chercher le prompt parfait. Et pourtant j'ai passé des heures à essayer de trouver la formule magique, la structure idéale, les mots-clés qui feraient que l'agent comprendrait enfin ce que je veux.

À la place, je me suis posé une question plus utile :

Dans quel système l'agent est-il en train de travailler et à quel contexte ce système appartient-il ?

Un développeur humain ne travaille jamais uniquement à partir d'une consigne. Il s'appuie sur tout un environnement : conventions du projet, tests disponibles, habitudes de l'équipe, historique des choix, zones sensibles, intuitions accumulées sur ce qu'il faut éviter de casser. J'ai la prétention d'être un développeur expérimenté, et pourtant je me suis allègrement assis sur des années de pratiques en traitant mon agent de codage comme un outil magique quasi omniscient et capable de deviner ce que je ne lui dis pas.

Le prompt, même long et détaillé, n'est qu'une petite partie de l'information disponible. La majeure partie du contexte se trouve ailleurs. Si tout doit passer par le prompt, celui-ci devient énorme, fragile, incomplet, et doit être reconstruit à chaque tâche. Si une partie de l'information vit dans le repo, sous une forme lisible, stable et versionnée, mais surtout accessible et référencée, alors le prompt peut redevenir ce qu'il devrait être : une consigne ciblée pour une tâche précise.

Ce qui est connu de l'équipe devient alors connu de l'agent, non pas parce qu'il le devine, mais parce que c'est « écrit quelque part ».

Ma formule mentale est devenue quelque chose comme :

```
Résultat fiable =
  contexte projet explicite (dans le repo)
  + tâche bornée (dans le prompt)
  + validation exécutable (commandes qu'il peut lancer)
  + permissions adaptées au risque
  + review humaine sur ce qui compte vraiment
```

L'IA n'a pas créé ces éléments, ils ne sont pas nouveaux et vous les utilisiez déjà avant que l'IA ne s'invite dans vos process de développement. C'est ce que l'on attend, avec ou sans IA, d'un workflow d'équipe sérieux et c'est ce sur quoi on m'a cassé les pieds quand j'étais junior et ce sur quoi j'ai cassé les pieds à tous les juniors avec qui j'ai travaillé après. La seule différence, c'est qu'il faut désormais le rendre lisible pour un agent : écrit, versionné, exécutable, plutôt que stocké uniquement dans la tête des seniors.

Le point des permissions mérite aussi d'être pris au sérieux. Sur une tâche documentaire, un refactoring local ou une exploration isolée, je peux laisser plus de latitude. Sur de l'authentification, du paiement, des migrations de données, de l'infrastructure ou du code exposé à un risque sécurité, je veux au contraire réduire la surface d'action : commandes autorisées, écriture limitée, validation explicite, review systématique.

Un agent de codage n'est pas dangereux parce qu'il est « intelligent ». Il devient dangereux quand on lui donne un périmètre flou avec des permissions trop larges.

L'équilibre dépend du contexte, du produit, de l'équipe et de l'appétit pour le risque. Certains mettront plus de poids sur la review humaine, d'autres sur le contexte explicite, d'autres encore sur l'isolation de l'environnement. L'important n'est pas la pondération exacte. L'important est d'arrêter de demander à un agent de deviner ce que l'équipe sait déjà et de le garder sur des rails clairs, avec des aiguillages et des points de contrôle réguliers.

D'une intention à une tâche d'ingénierie

Concrètement, ce qui a le plus changé pour moi au quotidien, c'est la façon dont je formule mes demandes.

Avant, j'écrivais des consignes du type :

```
Améliore la page de réglages utilisateur.
```

C'était rapide à écrire. C'était fluide. Mais cela laissait à l'agent le soin de décider ce que « améliorer » signifiait.

- Améliorer le design ?
- Réduire la duplication ?
- Changer le layout ?
- Extraire des composants ?
- Revoir la validation ?
- Modifier l'accessibilité ?

Tout cela, et bien d'autres choses encore, peuvent être des améliorations. Faire ce type de demande signifie simplement que je laisse à l'agent le soin de choisir laquelle de ces améliorations il va faire, et comment il va les faire. Je lui donne une intention, pas une tâche d'ingénierie.

Aujourd'hui, sur une tâche équivalente, j'écris plutôt :

```
Refactor la page de réglages utilisateur pour supprimer la duplication de la logique de
validation de formulaire.

Périmètre :
   tu ne peux modifier que les fichiers sous src/features/settings ;
   ne change pas les contrats API ;
   ne change pas le layout visuel.

Comportement attendu :
   les messages de validation existants restent identiques ;
   le comportement à la soumission reste identique ;
   les helpers de validation partagés sont extraits localement.

Validation :
   lance npm test settings ;
   lance npm run lint ;
   si une commande ne peut pas être exécutée, dis-le et explique pourquoi.
```

Ce n'est pas plus élégant. Ce n'est pas une formule magique. C'est simplement plus borné.

L'agent n'a plus à interpréter « améliorer ». Il sait quel comportement conserver, quel périmètre respecter, quels éléments ne pas toucher et comment vérifier son propre travail avant de me rendre le résultat. Bref on lui a mis un cadre.

Je ne fais pas cela pour toutes mes demandes. Pour une exploration rapide, un brouillon de script ou une question ponctuelle, je continue souvent à écrire deux lignes. Mais dès qu'il y a un repository, une équipe, une convention à respecter ou un risque de casser quelque chose, je passe à ce format.

La différence de qualité est nette et régulière, sans être spectaculaire. L'agent ne devient pas soudain parfait. Il fait simplement moins de choix hasardeux, car il a moins de choix à faire.

Et le temps que je « perds » à écrire un brief, je le récupère largement à la review.

C'est aussi là qu'apparaît une chose plus profonde : écrire un bon brief, c'est déjà faire une partie du travail d'ingénierie. J'ai souvent pesté contre les demandes vagues d'un PM ou d'un chef de projet. Un agent, lui, ne peste pas ; si je ne suis pas assez clair, il infère… Et de mon coté je dois m'appliquer la même discipline que celle que j'attends du PM ou du chef de projet : si je n'arrive pas à dire ce que je veux, ce qui ne doit pas bouger, quel comportement doit rester identique et comment je saurai que la tâche est terminée, c'est probablement que je n'ai pas encore vraiment réfléchi au problème.

Un agent de codage ne peut pas combler cela à ma place de façon sure. Il va combler, c'est certain, mais le résultat sera aléatoire.

Il peut produire vite. Il peut explorer. Il peut proposer. Mais il ne peut pas transformer une intention vague en décision produit ou architecture solide sans contexte. À un moment, la qualité de la demande reflète la qualité de la pensée en amont.

Le prompt ne compense pas un repo illisible

Si je devais retenir une seule chose de cette pratique, ce serait celle-là : un bon prompt sur un repo illisible donne un résultat fragile, alors qu'un prompt sobre sur un repo bien instrumenté donne souvent un résultat solide.

Cela signifie qu'à un moment, il faut investir dans le repo lui-même.

Pas en construisant une usine à gaz. Pas en écrivant une documentation interminable que personne ne lira. Simplement en rendant explicites les informations que l'on aurait de toute façon expliquées à un nouvel arrivant.

OpenAI recommande, pour Codex, de fournir le layout du projet, les commandes de build, de test et de lint, les conventions, les contraintes, les « do not » et ce que « done » signifie dans le projet.[2] Cette liste n'a rien d'une checklist administrative. C'est la documentation minimale qui aurait dû exister de toute façon, quel que soit l'agent que l'on branche ensuite sur le repository.

La plupart du temps, cela vit dans un fichier **AGENTS.md** à la racine du repo, plus quelques **AGENTS.md** locaux pour les zones qui ont leurs spécificités. La mécanique fera l'objet de l'article suivant ; ce qui compte ici, c'est l'idée.

Une convention qui n'est écrite nulle part est une convention qu'un agent finira par casser.

Et c'est normal.

Ce n'est pas lui faire un procès. C'est reconnaître qu'il ne peut pas respecter une règle qu'on ne lui donne pas. Si une équipe travaille avec des conventions implicites, elle peut s'en sortir entre humains, parce que les humains apprennent par friction sociale : remarques en review, discussions de couloir, habitudes, mémoire du projet. Un agent, lui, a besoin d'un support plus explicite.

Côté outils non-Codex, le principe est identique. Les noms changent (**CLAUDE.md** pour Claude Code, **.cursorrules** pour Cursor, ou d'autres fichiers selon les environnements) mais le geste reste le même : sortir l'implicite des têtes pour le poser dans le repo.

En bonus, un repo lisible pour un agent est souvent un repo plus lisible pour l'équipe.

C'est probablement l'un des effets secondaires les plus intéressants de ces outils : ils forcent à formaliser ce que l'on tolérait depuis des années sous forme de savoir oral.

Où l'agent travaille change ce qu'il peut faire

Autre chose que j'ai mis du temps à intégrer : la surface de travail change tout.

Un agent de codage peut travailler dans une app dédiée, dans une CLI, dans une extension IDE, en review de PR GitHub, ou dans un worktree pour isoler une tâche. Ce n'est pas un détail d'interface. C'est un choix de contrôle.

La surface conditionne la feedback loop, donc la qualité du résultat.

En exploration, je préfère la CLI ou l'IDE. Le feedback est court. Je peux interrompre, ajuster, relancer, lire le diff immédiatement, revenir à la main si nécessaire. C'est le bon mode pour comprendre, expérimenter ou corriger rapidement.

Pour une tâche bien bornée que je veux isoler de mon travail en cours, j'utilise plutôt un worktree. OpenAI documente ce pattern dans l'app Codex[3], et c'est probablement l'un des changements de workflow qui m'a fait gagner le plus de temps ces derniers mois. Je peux confier une tâche à un agent sans polluer mon checkout courant, puis revenir au résultat quand il est prêt à être relu.

Avant ma review humaine sur une PR un peu chargée, je laisse parfois un agent faire une review automatique. Codex propose ce type d'usage avec GitHub[4], mais le principe existe ailleurs sous d'autres formes. Cela ne remplace jamais ma review. Cela ne remplace pas non plus la responsabilité d'un développeur qui comprend le produit. Mais cela peut remonter des régressions potentielles, des tests manquants ou des incohérences que j'aurais pu manquer en lecture rapide.

La bonne question n'est donc pas :

```
Quelle surface est la meilleure ?
```

La bonne question est plutôt :

```
Quelle surface me donne la meilleure feedback loop pour cette tâche, avec un niveau de risque
que j'accepte ?
```

Pour un changement d'authentification, de paiement, de migration de données ou de sécurité, je veux du contrôle, de l'isolation et une review attentive.

Pour une mise à jour documentaire, une extraction locale ou un nettoyage de code bien borné, je veux surtout de la vitesse.

Même outil, postures très différentes.

Là encore, la logique est transposable. Les autres agents ont leurs propres surfaces : Claude Code en CLI, Cursor en IDE, Aider dans sa propre boucle de travail. Les détails diffèrent, l'arbitrage reste le même : choisir la surface qui donne le bon équilibre vitesse/contrôle pour la tâche en question.

Par où commencer

Pour qui voudrait expérimenter, voici trois angles d'attaque, dans l'ordre où ils m'ont le plus apporté.

Ce ne sont pas des étapes obligatoires. Ce sont trois pistes indépendantes, qui ne demandent ni infrastructure lourde ni outillage particulier.

1. Écrire un brief de cinq lignes avant la prochaine tâche non triviale.

Objectif, périmètre, ce qui ne doit pas changer, comportement attendu, méthode de validation. Pas plus.

Comparer ensuite le résultat avec votre façon habituelle de demander.

Chez moi, l'écart a été visible dès la première fois. Pas parce que l'agent devenait plus intelligent, mais parce que je lui retirais des choix qu'il n'aurait pas dû avoir à faire.

2. Créer un AGENTS.md minimal à la racine d'un repo.

Dix à quinze lignes maximum.

Structure du repo, commandes de validation, deux ou trois conventions importantes, deux ou trois « do not ». Surtout, ne pas chercher l'exhaustivité. C'est un fichier vivant, qui se complète à chaque fois que l'agent se trompe sur quelque chose. Il ne faut pas hésiter à lui dire « Tu t'es trompé ici, ajoute une règle dans le fichier d'instructions du projet pour que tu ne refasses pas la même erreur la prochaine fois. ».

C'est ainsi que je l'ai construit chez moi : à chaque fois que je devais reprendre la même explication, je la déplaçais dans le fichier.

Au bout de deux semaines, le fichier était utile sans avoir jamais été « conçu » comme une grande documentation projet.

3. Demander à l'agent une evidence note ou une ADR à la fin de chaque tâche.

Résumé du changement, fichiers modifiés, commandes lancées et résultats, commandes non lancées et pourquoi, points à regarder en review.

Cela change la façon dont on relit le diff ensuite. On passe d'une lecture en aveugle à une lecture guidée par l'agent lui-même. Cela ne dispense pas de lire le code, mais cela donne une première carte du terrain.

Accessoirement, cela permet de repérer rapidement les cas où l'agent affirme que tout va bien sans avoir réellement validé. Une evidence note incomplète, vague ou trop confiante est souvent un signal à prendre au sérieux.

Si après quelques essais cela n'apporte rien, autant passer à autre chose. Tous les workflows ne méritent pas d'être instrumentés. Mais si quelque chose s'en dégage, alors le harness, les workflows et la boucle qualité deviennent les sujets naturels à traiter ensuite.

Ce sera l'objet des articles suivants.

Et pour les autres agents de codage ?

Je parle de Codex parce que c'est l'outil que j'utilise et qui me convient aujourd'hui. Mais à peu près tout ce qui précède vaut aussi pour Claude Code, Gemini, Cursor, Aider, Copilot Workspace et les autres agents de codage.

L'idée d'agent de travail plutôt que de chatbot, le besoin d'un contexte projet explicite, les tâches bornées, la boucle de validation, la review humaine sur ce qui compte vraiment : rien de tout cela n'est spécifique à Codex.

Les noms de fichiers changent. Certaines surfaces n'existent pas partout. Les worktrees Codex ont des équivalents ailleurs, parfois moins intégrés, parfois plus artisanaux. La granularité des permissions ou les workflows varient selon les outils mais sont généralement présents d'une manière ou d'une autre.

Mais tant que ces agents travaillent dans des repositories réels, avec des conventions, des tests, des risques, des arbitrages humains et des contraintes produit, la logique reste la même.

Si vous utilisez autre chose que Codex, les exemples sont donc largement transposables. Gardez la substance, faites la traduction.

Ce qui reste humain

Reste ce qui, malgré tout cela, n'a pas changé.

Un agent peut produire vite, lancer des validations, relire un diff, proposer des corrections. Mais il ne sait pas pourquoi le produit existe. Il ne sait pas pourquoi telle décision d'architecture a été prise il y a six mois. Il ne sait pas ce qui fera râler les clients si on le casse. Il ne sait pas ce que l'on a promis à un partenaire la semaine dernière.

Tout cela reste du côté de l'humain.

C'est ce qui rend une décision technique défendable au-delà du code.

Mon travail n'a pas diminué ; il s'est déplacé.

Je passe moins de temps à écrire du code ligne par ligne, et plus de temps à concevoir le système dans lequel l'agent l'écrit. Le cadrage, le scope, les contraintes, la validation, la review : c'est là que se concentre désormais l'essentiel de la valeur.

Ce déplacement m'a finalement convenu mieux que je ne l'aurais prévu.

Il oblige à être plus explicite. Plus rigoureux. Moins tolérant avec les consignes floues et les repositories illisibles. Il ne supprime pas le travail d'ingénierie ; il le déplace vers l'amont et vers la boucle de contrôle.

Articles liés

- Construire un harness de projet : **AGENTS.md**, commandes de validation et limites de tâche
- Transformer une demande vague en tâche d’ingénierie vérifiable
- Fermer la boucle qualité : tests, review humaine et evidence notes

Références

1. « OpenAI Developers, Best practices — Codex », consulté le 2026-06-07.
https://developers.openai.com/codex/learn/best-practices

2. « OpenAI Developers, Best practices — Codex », sections sur contexte projet, validation et anti-patterns, consulté le 2026-06-07.
https://developers.openai.com/codex/learn/best-practices

3. « OpenAI Developers, Worktrees — Codex app », consulté le 2026-06-07.
https://developers.openai.com/codex/app/worktrees

4. « OpenAI Developers, Codex code review for GitHub pull requests », consulté le 2026-06-07.
https://developers.openai.com/codex/use-cases/github-code-reviews