Notilis Tech

Fermer la boucle qualité : tests, review humaine et evidence notes

Fermer la boucle qualité avec tests, review humaine, boucle de retour et evidence notes.

2026-05-24 · Tests · Review · Evidence

Résumé

La qualité d'un changement assisté par IA ne se vérifie pas seulement à la fin. Elle se construit pendant toute la boucle : cadrage, modification, validation, correction et trace.

Avec un agent de codage, un diff peut arriver vite, mais la review humaine ne doit pas devenir le seul endroit où l'on découvre l'intention, les limites et les validations manquantes.

Cet article propose une boucle qualité simple : tests au bon moment, séparation entre review agent et review humaine, et evidence notes pour garder les preuves hors du chat fog. J'utilise Codex comme cas d'étude, parce que c'est mon outil quotidien, mais la logique s'applique largement aux autres agents de codage.

Sommaire

Contexte de lecture

Audience

Cet article s'adresse aux développeurs, QA automation, tech leads et reviewers qui veulent garder une boucle qualité solide avec des changements assistés par IA.

Ce que cet article couvre

Il détaille la place des tests, de la review humaine, de la review agent et des evidence notes dans une boucle de retour vérifiable, avec Codex comme exemple principal.

Ce que cet article ne couvre pas

Il ne remplace pas une stratégie de test complète et ne prétend pas que l'IA peut garantir seule la qualité d'un changement.

Le test n’est pas une formalité finale

Dans un workflow classique, les tests sont souvent exécutés vers la fin : une fois le code écrit, on lance la suite, on corrige ce qui casse, puis on prépare la PR. Avec un agent de codage, il faut déplacer la validation plus tôt dans la boucle, parce que l’agent peut produire très vite un volume de modifications suffisant pour rendre la correction plus coûteuse si l’erreur est découverte tard.

La boucle de retour idéale dépend de la tâche, mais le principe reste le même : une hypothèse doit être vérifiée dès que possible.

Pour un bug reproductible, la boucle peut être :

```
reproduire le bug
-> ajouter un test en échec ou documenter la reproduction
-> implémenter la correction minimale
-> lancer le test ciblé
-> lancer une validation élargie
-> produire une evidence note
```

Pour un refactoring local :

```
inspecter le comportement existant
-> identifier les tests stables
-> faire une petite modification
-> lancer les tests ciblés
-> inspecter le diff
-> lancer lint/build si pertinent
-> documenter le comportement inchangé
```

Pour une documentation technique :

```
inspecter le comportement source
-> mettre à jour la documentation
-> vérifier les exemples ou les commandes
-> croiser les références
-> noter les hypothèses
```

Le point important est qu’un agent ne doit pas seulement produire un diff. Il doit participer à la validation du diff. OpenAI recommande de donner à Codex des indications sur les commandes de build, test et lint, ainsi que sur ce que « terminé » signifie dans le projet.[1] Ces informations doivent donc être dans le harness et reprises dans la tâche.

Exemple concret : corriger un bug de filtre sans perdre la boucle qualité

Prenons un bug volontairement simple.

Une API expose une route de recherche de commandes :

```
GET /api/orders?status=CLOSED
```

Le comportement attendu est simple : seules les commandes fermées doivent être retournées.

Le bug observé est le suivant : lorsque le paramètre **search** est absent, le filtre **status** est ignoré. L'API retourne alors des commandes **OPEN**, **PENDING** et **CLOSED**.

Le risque est typique d'un bug assisté par IA : un agent peut corriger rapidement le fichier suspect, mais si on ne lui impose pas une boucle de validation, il peut produire une correction plausible sans test de régression, ou modifier trop largement la logique de recherche.

Je préfère généralement travailler en deux passes : une passe d'isolation et une passe de correction. Cela permet au prompt de correction d'être plus direct, en se concentrant uniquement sur la cause du bug, sans devoir aussi prouver la reproduction.

Le task brief donné à l'agent peut ressembler à ceci :

```
Contexte :
Le projet contient une API Node/TypeScript pour gérer des commandes.
La route `GET /api/orders` accepte notamment les paramètres `status` et `search`.

Bug observé :
Quand on appelle `GET /api/orders?status=CLOSED` sans paramètre `search`, l'API retourne aussi
des commandes dont le statut n'est pas `CLOSED`.

Comportement attendu :
Le filtre `status` doit être appliqué même lorsque `search` est absent ou vide.

Objectif :
Reproduire le bug avec un test de régression, corriger le bug de façon minimale, puis valider
la correction.

Contraintes :
- Ne pas refactorer globalement le module de recherche.
- Ne pas modifier le contrat public de l'API.
- Ne pas changer les noms des statuts existants.
- Ne pas ajouter de dépendance.
- Commencer par identifier le fichier responsable.
- Ajouter ou modifier un test ciblé qui échoue avant correction.
- Ne pas corriger maintenant la cause du bug.

Méthodologie :
Tu dois travailler en deux passes :
- La première passe doit se concentrer sur la reproduction du bug et la validation de la
reproduction.
- La deuxième passe doit se concentrer sur la correction du bug et la validation de la
correction.
- Arrête-toi après la première passe et renvoie un rapport d'analyse même si tu trouves la
cause du bug, pour laisser la validation humaine prouver le bug avant de le corriger.

Résultat intermédiaire attendu :
Un rapport d'analyse incluant
- les fichiers inspectés,
- les tests ajoutés,
- les résultats des tests,
- hypothèse de root cause.

Résultat final attendu :
- Un test de régression couvrant `status` sans `search`.
- Une evidence note listant les fichiers modifiés, les commandes exécutées, les validations
non exécutées et les risques restants.
```

Le résultat intermédiaire attendu n'est pas encore une correction. Le premier résultat attendu est plutôt quelque chose comme :

```
Reproduction :
- Fichier inspecté : `src/orders/order-query.ts`
- Test ajouté : `tests/orders/order-query.test.ts`
- Test ciblé exécuté :
  `npm test -- tests/orders/order-query.test.ts -t "applies status filter without search"`
- Résultat avant correction :
  échec confirmé, le résultat contient des commandes `OPEN` alors que le filtre demandé est
`CLOSED`.

Hypothèse :
Le filtre `status` est appliqué uniquement dans une branche conditionnelle liée à `search`.
```

Cela force l'agent à prouver le bug avant de le corriger. C'est important : la reproduction devient une preuve, pas une narration.

Un exemple simplifié du bug dans le code pourrait être celui-ci :

```
export function buildOrderWhere(filters: OrderFilters): OrderWhereInput {
  const where: OrderWhereInput = {};

  if (filters.search) {
    where.OR = [
      { reference: { contains: filters.search } },
      { customerName: { contains: filters.search } },
    ];

    if (filters.status) {
      where.status = filters.status;
    }
  }

  return where;
}
```

Ici, le problème est visible : **status** est appliqué seulement si **search** existe. La correction minimale consiste à sortir le filtre **status** de la condition liée à **search**.

```
export function buildOrderWhere(filters: OrderFilters): OrderWhereInput {
  const where: OrderWhereInput = {};

  if (filters.status) {
    where.status = filters.status;
  }

  if (filters.search) {
    where.OR = [
      { reference: { contains: filters.search } },
      { customerName: { contains: filters.search } },
    ];
  }

  return where;
}
```

Le test de régression attendu pourrait ressembler à ceci :

```
it("applies the status filter when search is empty", async () => {
  const result = await listOrders({
    status: "CLOSED",
    search: "",
  });

  expect(result.items.length).toBeGreaterThan(0);
  expect(result.items.every((order) => order.status === "CLOSED")).toBe(true);
});
```

Une fois la cause identifiée sans ambiguité et validée par une relecture humaine du code, un second prompt de correction peut être lancé, avec comme résultat attendu la correction du bug et la validation de la correction. Par exemple :

```
Le test de régression confirme que le filtre `status` est ignoré lorsque `search` est absent
ou vide.

Corrige maintenant la cause du bug.

Contraintes :
- Correction minimale.
- Aucun refactoring global.
- Ne modifie pas le contrat de `GET /api/orders`.
- Ne change pas les fixtures sauf nécessité prouvée.
- Après correction, relance le test ciblé.
- Si le test passe, lance la suite du module orders.
- Termine par une evidence note.
```

Le résultat final attendu de l'agent devrait ensuite être de ce type :

```
Résumé :
Correction du filtrage des commandes pour appliquer `status` indépendamment du paramètre
`search`.

Fichiers modifiés :
- `src/orders/order-query.ts`
- `tests/orders/order-query.test.ts`

Ce qui a changé :
- Le filtre `status` est maintenant appliqué avant la construction de la condition de
recherche texte.
- Le comportement de `search` reste inchangé.
- Un test de régression couvre le cas `status=CLOSED` sans recherche texte.

Validations exécutées :
- `npm test -- tests/orders/order-query.test.ts -t "applies the status filter when search is
empty"` : passé.
- `npm test -- tests/orders/order-query.test.ts` : passé.
- `npm run lint -- src/orders/order-query.ts tests/orders/order-query.test.ts` : passé.

Validations non exécutées :
- `npm test` complet non exécuté, car la suite complète dépasse le temps disponible dans
l'environnement local.
- Recommandation : lancer la suite complète en CI avant merge.

Vérification du scope :
- Aucun changement du contrat public de l'API.
- Aucun changement des statuts existants.
- Aucun changement de pagination.
- Aucun changement du comportement de recherche texte.

Limites connues :
- La correction ne couvre pas les autres filtres éventuels.
- La validation E2E de la route HTTP complète reste à confirmer si elle existe dans le projet.

Points d'attention pour la review :
- Vérifier que l'ordre d'application de `status` et `search` correspond bien au comportement
attendu.
- Vérifier que le test utilise des données représentatives.
- Vérifier que le bug ne se répète pas sur d'autres filtres comme `customerId`, `dateFrom` ou
`dateTo`.
```

Avant la review humaine, j'essaye généralement de faire une review agent.

```
Relis le diff comme review agent.

Objectif :
Chercher les risques mécaniques, les tests manquants et les incohérences locales.

Ne propose pas de refactoring cosmétique.
Ne valide pas l'intention produit.
Ne décide pas si le changement doit être mergé.

Vérifie uniquement :
- Le test reproduit-il bien le bug ?
- La correction est-elle minimale ?
- Le filtre `status` fonctionne-t-il avec et sans `search` ?
- Le comportement de `search` reste-t-il inchangé ?
- Un autre filtre semble-t-il affecté ?
- Les validations annoncées correspondent-elles au diff ?
```

Le résultat attendu de cette review agent n'est pas « approved ». Il doit plutôt ressembler à :

```
Review agent :

Signal principal :
- La correction est locale et cohérente avec le bug décrit.
- Le test couvre bien le cas `status` sans `search`.

Points à vérifier humainement :
- Le test utilise-t-il les mêmes conventions de fixtures que le reste du module ?
- Faut-il ajouter un second test combinant `status=CLOSED` et `search=...` pour éviter une
régression sur la composition des filtres ?

Risque faible :
- Aucun changement visible du contrat API.
- Aucun changement de pagination ou de tri.

Limite :
- La review ne confirme pas l’intention produit sur les combinaisons de filtres avancées.
```

Tout ceci n'est évidemment qu'un exemple extrêmement simple et si vous avez besoin d'utiliser un agent de codage pour isoler ce bug et le fixer, évitez de trop le dire à votre tech lead ou à d'autres membres de l'équipe .

Ensuite, tout ça doit être adapté au contexte du projet, à la nature du bug, à la criticité du changement et à la culture de l’équipe. L’objectif est de montrer comment construire une boucle qualité avec un agent de codage. Codex sert ici de cas d’étude, pas de recette universelle.

Validation ciblée, puis validation élargie

Toutes les validations ne doivent pas être lancées au même moment. Une bonne boucle commence souvent par une validation ciblée, puis s’élargit.

La validation ciblée répond à la question : le changement demandé fonctionne-t-il là où il doit fonctionner ? Elle utilise un test de module, un test de composant, une commande de scénario de reproduction, une petite suite ciblée ou un smoke test local.

La validation élargie répond à la question : le changement a-t-il cassé quelque chose autour ? Elle utilise un lint global, un build, une suite plus large, une vérification de types, ou un test E2E selon le projet.

Un exemple de chemin de validation peut être :

```
Chemin de validation :
1. Lancer le test de régression ciblé.
2. Lancer la suite de tests du module concerné.
3. Lancer le lint sur le package modifié.
4. Lancer le build complet seulement si les checks précédents passent.
5. Si une commande échoue, classer l’échec comme lié au changement, préexistant ou inconnu.
```

Cette structure évite deux extrêmes. Le premier consiste à ne rien valider. Le second consiste à lancer une suite énorme trop tôt, sans savoir si le changement ciblé fonctionne déjà. La validation doit être proportionnée.

Review agent et review humaine ne vérifient pas la même chose

Il est tentant de dire qu'un agent peut reviewer le code, ou au contraire qu'une review agent ne vaut rien. Les deux positions sont trop simples.

Une review agent peut être utile pour chercher certains types de problèmes : régressions évidentes, tests manquants, incohérences locales, erreurs de documentation, oubli d'un cas, complexité inutile, ou surface de risque spécifique. OpenAI présente par exemple un use case de Codex code review pour GitHub pull requests, destiné à faire remonter des régressions potentielles, tests manquants et problèmes de documentation avant la review humaine.[2]

Mais cette review ne remplace pas le jugement humain. Le reviewer humain garde les arbitrages d'architecture, la compréhension du produit, les contraintes métier, l'acceptation du risque, les compromis de maintenance et la décision finale de merge.

La bonne séparation est donc :

Type de vérification	Review agent	Review humaine
Tests manquants évidents	Forte utilité	Vérification finale
Régression locale plausible	Forte utilité	Confirmation
Style et conventions simples	Utilité moyenne	Arbitrage selon contexte
Architecture	Assistance possible	Responsabilité humaine
Product intent	Faible sans contexte	Responsabilité humaine
Sécurité critique	Signal complémentaire	Review renforcée
Contrats API	Assistance possible	Validation humaine
Risk acceptance	Non	Responsabilité humaine

Cette distinction permet d'éviter deux erreurs. La première consiste à supprimer la review humaine parce que l'agent a relu. La seconde consiste à ignorer les signaux agent alors qu'ils peuvent faire gagner du temps sur des vérifications mécaniques.

Une review humaine orientée risque

Avec un agent de codage, la review humaine doit évoluer. Elle ne devrait pas seulement relire ligne par ligne pour voir si le code « semble correct ». Elle doit vérifier l'intention, le scope, les contrats, les risques et les preuves.

Une review efficace peut suivre cinq questions :

```
1. Le diff respecte-t-il le scope ?
2. Les non-goals sont-ils respectés ?
3. Le comportement attendu est-il prouvé ?
4. Les validations exécutées sont-elles suffisantes ?
5. Les risques connus sont-ils documentés ?
```

Cette review est plus proche d'un audit de changement que d'une simple lecture de code. Elle est particulièrement importante parce qu'un agent peut produire une solution plausible qui manque une

contrainte implicite.

Le reviewer doit donc utiliser le task brief comme référence. Si le task brief était mauvais, la review doit le noter. Sinon, l'équipe risque de corriger le diff sans corriger le système qui l'a produit.

Evidence notes : sortir les décisions du chat fog

Le chat est un mauvais système de traçabilité. Les décisions y sont dispersées, les validations y sont mélangées aux hypothèses, et les limites connues finissent par disparaître dans le fil de conversation. Pour une tâche sérieuse, il faut une evidence note.

Une evidence note est un court artefact qui résume ce qui a été fait, ce qui a été validé, ce qui n'a pas pu être validé, quels fichiers ont changé, quels risques restent ouverts, et quelles décisions ont été prises.

Elle peut être incluse dans la réponse finale de l'agent, dans la description de PR, dans un fichier temporaire, ou dans un commentaire de review. L'emplacement importe moins que la discipline : la preuve doit être accessible hors du chat.

Un template simple :

```
# Evidence note

## Tâche

Description courte du changement demandé.

## Fichiers modifiés

- `path/to/file`
- `path/to/test`

## Ce qui a changé

Explication courte de l'implémentation.

## Validations exécutées

- `commande` : résultat
- `commande` : résultat

## Validations non exécutées

Lister les commandes non exécutées et expliquer pourquoi.

## Vérification du scope

Confirmer ce qui est resté inchangé.

## Limites connues

Risques connus, hypothèses ou tâches de suivi.

## Points d'attention pour la review

Ce que le reviewer humain doit inspecter.
```

Cette note force l'agent à rendre son travail vérifiable. Elle aide aussi le reviewer à ne pas relire à l'aveugle.

Les validations non exécutées doivent être visibles

Il est normal que certaines validations ne puissent pas être exécutées : dépendance absente, environnement incomplet, test trop long, accès réseau indisponible, secret non disponible, base de données non démarrée. Le problème n'est pas toujours l'absence d'exécution. Le problème est l'absence de visibilité.

Une evidence note doit distinguer trois cas :

Cas	Ce qu'il faut écrire
Commande exécutée et passée	Commande + résultat.
Commande exécutée et échouée	Commande + erreur + classification.
Commande non exécutée	Raison + vérification manuelle recommandée.

Cette distinction évite les fausses assurances. Un diff non testé peut être acceptable dans certains cas, mais il ne doit pas être présenté comme validé.

Boucle de correction

Une bonne boucle qualité inclut le droit de corriger. Il ne faut pas considérer le premier diff comme le résultat final. L'agent doit pouvoir utiliser les retours de tests, les erreurs de lint, les commentaires de review et les observations humaines pour itérer.

Une boucle de correction peut suivre ce schéma :

```
Diff initial
-> validation ciblée
-> analyse de l'échec
-> correction minimale
-> relancer la validation ciblée
-> validation élargie
-> evidence note
-> review humaine
```

La règle importante est de garder les corrections minimales. Si un test échoue, l'agent ne doit pas partir dans un refactoring global sauf si la cause le justifie. Chaque correction doit rester reliée à la tâche.

Matrice de review

Une matrice simple peut aider l'équipe à décider quoi vérifier selon le type de changement :

Type de changement	Validation agent attendue	Review humaine attendue
Documentation	Vérification des exemples, liens et cohérence avec code	Clarté, exactitude, public cible
Test ajouté	Exécution du test ciblé	Pertinence du cas couvert

Type de changement	Validation agent attendue	Review humaine attendue
Bugfix local	Scénario de reproduction, test de régression, tests ciblés	Cause réelle, effets de bord
Refactoring local	Tests existants, lint, diff minimal	Lisibilité, maintien des contrats
Changement API	Tests contrat, build, documentation	Compatibilité, versioning, impact client
Sécurité / auth	Tests ciblés, recherche de régressions	Review senior, threat model, acceptation du risque
Migration	Dry-run, tests migration, rollback notes	Plan opérationnel, compatibilité, données

Cette matrice n’a pas besoin d’être parfaite. Elle sert à aligner les attentes avant que la tâche commence.

Ce que l’agent doit rendre à la fin

La réponse finale d’un agent de codage ne devrait pas se limiter à « terminé ». Elle devrait contenir au minimum : résumé du changement, fichiers modifiés, validations exécutées, validations non exécutées, limites connues et points d’attention pour la review recommandés. Le meilleur moyen pour l'obtenir est de garder le template de réponse dans un dossier et de préciser dans l'**AGENTS.md** quand utiliser ce template. Cette recommandation vaut aussi pour les autres réponses qui exigent de suivre une forme reproductible : leurs résultats doivent être structurés et ne doivent pas se limiter à un simple signal « terminé ».

Un format utile :

```
Résumé :
- ...

Fichiers modifiés :
- ...

Validation :
- `...` : passé
- `...` : échoué parce que ...
- Non exécuté : ... parce que ...

Limites connues :
- ...

Points d’attention pour la review :
- ...
```

Ce format est simple, mais il change la review. Le développeur humain n’est plus obligé de reconstruire tout le parcours. Il peut vérifier les points importants.

Les erreurs fréquentes

- La première erreur consiste à valider seulement à la fin. Plus l’erreur est découverte tard, plus elle coûte cher.

- La deuxième erreur consiste à croire qu'une review agent remplace la review humaine. Elle donne un signal supplémentaire, pas une décision de merge.
- La troisième erreur consiste à accepter une tâche sans chemin de validation. Une tâche non validable doit être re-cadrée ou traitée comme exploration.
- La quatrième erreur consiste à laisser les résultats de test dans le chat. Ils doivent être repris dans une evidence note ou dans la PR.
- La cinquième erreur consiste à ignorer les tests non exécutés. Ils doivent être explicitement listés.

Checklist opérationnelle

- [ ] Définir un chemin de validation avant la modification.
- [ ] Exécuter les checks ciblés dès que possible pendant la modification.
- [ ] Élargir la validation selon le risque après la modification.
- [ ] Produire une evidence note avant la review humaine.
- [ ] Vérifier scope, non-goals, contracts, validation et risques pendant la review humaine.

Cette boucle n'élimine pas les erreurs, mais elle les rend plus visibles et plus tôt.

Conclusion

La qualité d'un changement assisté par agent ne se décrète pas. Elle se construit dans une boucle : tâche claire, validation ciblée, correction, validation élargie, evidence note et review humaine orientée risque.

Codex peut aider à produire, tester, relire et corriger ; d'autres agents de codage peuvent entrer dans une boucle similaire avec leurs propres contraintes. Mais le développeur garde la responsabilité de la cohérence, des arbitrages et de l'acceptation du risque. La bonne approche n'est donc ni confiance aveugle, ni rejet systématique. C'est une boucle qualité structurée où chaque signal est utilisé à sa place.

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
- Industrialiser les workflows : Markdown avant frameworks, skills avant automations

Références

1. OpenAI Developers, « Best practices — Codex », section « Improve reliability with testing and review », consulté le 2026-06-07.
https://developers.openai.com/codex/learn/best-practices

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