Notilis Tech

Construire un harness de projet : AGENTS.md, commandes de validation et limites de tâche

Construire un harness de projet avec instructions projet, commandes de validation et limites de tâche pour rendre un agent de codage plus fiable.

2026-05-10 · Harness · Validation · AGENTS.md

Résumé

Le vrai levier de productivité avec un agent de codage n'est pas seulement la qualité du prompt. C'est la qualité du harness de projet : les règles, contraintes et validations que le dépôt rend explicites avant même qu'une tâche commence.

Cet article montre comment construire ce socle avec AGENTS.md, des commandes de validation et des limites de tâche. L'objectif n'est pas d'ajouter de la bureaucratie, mais de donner au travail assisté par IA des garanties d'ingénierie simples, lisibles et vérifiables.

Sommaire

Contexte de lecture

Audience

Cet article s’adresse aux développeurs, tech leads, platform engineers et engineering managers qui veulent rendre leur dépôt plus exploitable par un agent de codage.

Ce que cet article couvre

Il décrit les éléments d’un harness minimal : AGENTS.md, commandes de validation, limites, permissions, task brief, revue et artefacts de preuve.

Ce que cet article ne couvre pas

Il ne propose pas une plateforme complète d’orchestration ni une configuration universelle applicable à tous les projets.

Pourquoi un harness est nécessaire

Un dépôt sans harness force l'agent à inférer les règles à partir du code existant. Parfois cela fonctionne. Souvent, cela fonctionne partiellement. L'agent repère certains patterns, mais pas les exceptions. Il détecte une convention, mais pas son contexte. Il trouve une commande de test, mais ne sait pas si elle est suffisante. Il modifie une zone logique, mais ne sait pas qu'elle est historiquement fragile.

C'est exactement ce que le harness doit corriger. Il transforme les règles implicites en signaux explicites.

Le harness n'est pas un document unique. C'est un système léger composé de plusieurs éléments : des instructions projet, une configuration adaptée, des commandes de validation, des templates de task brief, des règles de permissions, des workflows de revue et des artefacts de preuve.

Son objectif est simple :

```
Réduire ce que l'agent doit deviner.
```

Plus l'agent doit deviner, plus le coût d'ambiguïté augmente. Plus le coût d'ambiguïté augmente, plus la revue humaine devient lourde. Le harness ne supprime pas cette revue, mais il la rend plus ciblée.

AGENTS.md comme contrat de collaboration

AGENTS.md est l'un des éléments les plus importants du harness, parce qu'il permet d'écrire les règles durables du projet. La documentation OpenAI décrit **AGENTS.md** comme un mécanisme d'instructions personnalisées pour Codex, avec une logique de fichiers d'instructions globaux, d'instructions au niveau du dépôt et éventuellement de surcharges dans des sous-dossiers.[1]

J'utilise ici Codex comme exemple principal, parce que c'est l'outil que j'utilise au quotidien et parce que sa documentation formalise bien ces mécanismes. Mais le principe du harness n'est pas spécifique à Codex. Claude Code, Gemini, Cursor, Aider, Copilot Workspace ou d'autres agents utilisent parfois d'autres fichiers, d'autres conventions et d'autres surfaces, mais le besoin reste le même : rendre le contexte du projet lisible, borné et vérifiable.

L'idée n'est pas de remplir **AGENTS.md** avec tout ce que l'équipe sait. L'idée est d'y placer ce que l'agent doit savoir régulièrement pour travailler correctement.

Un bon **AGENTS.md** répond à des questions concrètes : comment le projet est organisé, quelles commandes lancer, quelles conventions respecter, quels fichiers éviter, quelles dépendances ne pas ajouter, comment ouvrir une PR, et ce que "done" signifie.

Un exemple de structure peut ressembler à ceci :

```
# AGENTS.md

## Vue d'ensemble du dépôt

Ce dépôt contient l'API backend, l'application frontend et l'outillage partagé.

## Répertoires clés

- `backend/` : API et logique métier.
- `frontend/` : interface utilisateur.
- `docs/` : notes d'architecture et documentation opérationnelle.
- `scripts/` : utilitaires locaux pour les développeurs.

## Commandes de validation

- Tests backend : `cd backend && pytest`
- Tests frontend : `cd frontend && npm test`
- Lint frontend : `cd frontend && npm run lint`
- Smoke check complet : `./scripts/smoke-check.sh`

## Règles d'ingénierie

- Garder la logique métier hors des contrôleurs HTTP.
- Préférer les patterns existants de services et de sélecteurs.
- Ne pas introduire de nouvelles dépendances de production sans approbation explicite.
- Ne pas modifier les fichiers générés sauf si la tâche le demande explicitement.

## Critères de done

Une tâche est done seulement lorsque le comportement demandé est implémenté, que les tests
pertinents passent, et que toute commande de validation non exécutée est expliquée.
```

Ce document n'a pas besoin d'être parfait au premier jour. Il doit être utile, maintenu et corrigé après les erreurs observées. Chaque fois qu'un agent fait une mauvaise hypothèse récurrente, la question devrait être : faut-il améliorer **AGENTS.md**, le task brief, le workflow ou la validation ?

Instructions globales, projet et locales

La documentation OpenAI décrit une logique de chargement des instructions qui peut inclure des réglages par défaut globaux dans le répertoire personnel Codex, des instructions au niveau du dépôt, et des surcharges dans des répertoires spécialisés.[2] Cette logique est importante pour les équipes parce qu'elle permet de distinguer plusieurs niveaux de règles.

Les règles globales doivent rester personnelles ou transverses : préférence de style de travail, habitudes générales de validation, règles de prudence génériques. Les règles au niveau du dépôt doivent porter sur le projet : architecture, commandes, conventions, do-not rules. Les règles locales doivent porter sur des zones spécifiques : un service de paiement, un module legacy, une API critique, un package partagé.

Cette séparation évite deux erreurs. La première consiste à répéter partout les mêmes règles générales. La seconde consiste à mélanger dans un seul document des règles globales, des contraintes projet et des exceptions locales.

Pour un projet professionnel, un **AGENTS.md** au niveau du dépôt est souvent le point de départ le plus rentable. Il force l'équipe à expliciter les règles qui étaient jusque-là seulement connues des seniors.

La configuration outil ne remplace pas les instructions projet

La configuration est une autre partie du harness, mais elle ne joue pas le même rôle que **AGENTS.md**. OpenAI documente notamment un fichier de configuration utilisateur **~/.codex/config.toml** et des fichiers **.codex/config.toml** propres au projet, avec une logique de priorité entre flags CLI, profils, config projet, config utilisateur, config système et réglages par défaut.[3]

La configuration sert à adapter le comportement de l'outil : modèle, sandbox, approvals, profils, MCP, noms de fichiers de repli, ou autres options selon le contexte. Les instructions projet servent à expliquer comment l'équipe travaille.

Il faut donc éviter deux confusions.

- La première consiste à mettre des règles de travail dans la configuration alors qu'elles devraient être lues comme des instructions projet.
- La seconde consiste à mettre des réglages techniques dans **AGENTS.md** alors qu'ils devraient être configurés proprement.

La bonne séparation est la suivante :

Besoin	Emplacement naturel
Conventions d'architecture	AGENTS.md
Commandes de test et lint	AGENTS.md
Do-not rules projet	AGENTS.md
Sandbox, approvals, profils	configuration de l'outil
Noms de fichiers de repli pour les instructions	configuration de l'outil
Règles locales par module	AGENTS.md ou AGENTS.override.md local
Procédure récurrente complexe	workflow Markdown ou skill

Cette séparation rend le système plus maintenable. Elle permet aussi à l'équipe de relire les règles métier sans devoir interpréter un fichier de configuration.

Commandes de validation : le langage commun du done

Une tâche confiée à un agent sans commande de validation reste une tâche jugée à l'impression. Le résultat peut sembler correct, mais le feedback loop reste faible. L'agent a besoin de savoir comment vérifier son travail ; le développeur a besoin d'un signal pour relire autre chose que du texte généré.

Les commandes de validation sont donc le langage commun du done. Elles peuvent être simples : tests unitaires, lint, vérification de types, build, smoke test, reproduction d'un bug. Elles peuvent aussi être spécialisées : tests d'un module, script de migration dry-run, test E2E ciblé, requête de vérification,

commande de génération suivie d'un diff.

Le point essentiel est que la validation doit être explicitement reliée à la tâche. Demander "run tests" est parfois trop vague. Demander "run **pytest tests/test_invoice_service.py** and **npm run lint**" est plus utile. Si la commande ne peut pas être exécutée, l'agent doit expliquer pourquoi, au lieu de laisser un faux sentiment de validation.

Un bon task brief peut donc contenir une section dédiée :

```
Validation:
- Exécuter `pytest tests/test_user_settings.py`.
- Exécuter `npm run lint`.
- Si une commande échoue, expliquer si l'échec vient de ce changement ou d'un problème
préexistant.
- Si une commande ne peut pas être exécutée, indiquer pourquoi et ce qui doit être vérifié
manuellement.
```

Cette discipline évite une phrase trop fréquente : "I did not run tests". Cette phrase n'est pas toujours problématique, mais elle doit être une exception expliquée, pas le mode normal de validation.

Les limites de tâche réduisent le blast radius

Le blast radius est l'étendue des effets possibles d'un changement. Plus une tâche est vague, plus le blast radius augmente. L'agent peut toucher plus de fichiers, modifier plus de comportements, introduire plus d'hypothèses et rendre la revue plus coûteuse.

Les limites de tâche servent à réduire ce risque. Elles précisent ce qui est dans le scope, ce qui est hors scope, quels fichiers peuvent être modifiés, quels contrats ne doivent pas changer, quelles dépendances sont interdites, et quel niveau de refactoring est acceptable.

Une limite de tâche utile peut être écrite ainsi :

```
Scope:
- Tu peux modifier `backend/orders/services.py`.
- Tu peux ajouter des tests dans `backend/orders/tests/`.
- Ne pas modifier les serializers API.
- Ne pas modifier le schéma de base de données.
- Ne pas ajouter de dépendances.
- Ne pas refactorer la logique de commande non liée à cette tâche.
```

Ce niveau de contrainte peut sembler restrictif. En réalité, il améliore souvent la productivité. Une tâche plus petite produit un feedback loop plus court, une revue plus ciblée, et moins de surprises.

La règle pratique est simple :

```
Plus le risque est élevé, plus la limite doit être explicite.
```

Un changement de documentation peut accepter un scope plus souple. Un changement de paiement, d'authentification, de droits ou de données doit être borné beaucoup plus strictement.

Permissions et sandbox : la frontière opérationnelle

Les limites décrivent le scope dans le task brief. Les permissions et le sandbox définissent ce que l'agent peut réellement faire dans l'environnement. Ces deux niveaux doivent être cohérents.

Un task brief peut dire "do not modify files outside **src/**", mais si l'agent a accès complet, le contrôle repose encore beaucoup sur la consigne. À l'inverse, un sandbox plus strict peut empêcher certaines actions accidentelles, mais il ne remplace pas le scope écrit.

OpenAI souligne dans ses best practices plusieurs anti-patterns, dont le fait de donner des permissions complètes à l'ordinateur avant de comprendre le workflow, ou de transformer trop tôt une tâche récurrente en automatisation.[4] Cette prudence vaut particulièrement pour les équipes. Le niveau de permission doit suivre la maturité du workflow, pas l'impatience de gagner du temps.

Un workflow local et interactif peut commencer avec des permissions limitées, puis demander approbation pour les actions plus sensibles. Une automatisation doit être beaucoup plus stricte, car elle peut exécuter des actions sans supervision immédiate. Un workflow critique doit probablement imposer des approvals, des tests et une revue senior.

Le harness minimal viable

Il n'est pas nécessaire de tout construire avant de commencer. Un harness minimal viable peut tenir en quelques artefacts :

Artefact	Rôle
AGENTS.md	Instructions durables du dépôt.
Checklist de validation	Commandes de test, lint, build, smoke test.
Task brief template	Format standard des demandes à un agent.
Matrice de revue	Répartition revue agent / revue humaine.
Evidence note template	Trace des commandes, résultats, fichiers et limites.
Politique de limites	Règles de scope selon criticité.

Ce socle suffit déjà à changer la qualité du travail. Il transforme l'agent d'un outil opportuniste en participant d'un workflow lisible.

Template de task brief borné

Un template simple peut être utilisé par toute l'équipe :

```
# Task brief agent de codage

## Objectif

Décrire le comportement concret ou le changement attendu.

## Contexte

Expliquer pourquoi ce changement est nécessaire et où il s'inscrit dans le projet.

## Scope

Fichiers, dossiers ou modules que l'agent peut inspecter ou modifier.

## Non-goals

Ce qui ne doit pas être changé.

## Contraintes

Règles d'architecture, limites de dépendances, contrats API, enjeux de sécurité.

## Comportement attendu

Comportement observable après le changement.

## Commandes de validation

Commandes que l'agent doit exécuter avant d'annoncer que la tâche est terminée.

## Focus de revue

Ce que la personne qui relit doit regarder en priorité.

## Evidence attendue

Commandes exécutées, résultats des tests, fichiers modifiés, limites connues.
```

Ce template n'est pas un formulaire bureaucratique. Il sert à éviter que l'agent commence à écrire avant que la tâche soit réellement comprise.

Les erreurs fréquentes

- La première erreur consiste à écrire **AGENTS.md** comme un document de doctrine trop général. Un agent a besoin de règles actionnables, pas seulement de principes.
- La deuxième erreur consiste à mettre dans chaque prompt des informations qui devraient être durables. Si une règle revient souvent, elle doit probablement être dans **AGENTS.md**, un workflow ou une skill.
- La troisième erreur consiste à oublier les commandes de validation. Sans validation, la tâche reste subjective.
- La quatrième erreur consiste à ne pas limiter les fichiers. Une demande large peut produire un diff impressionnant, mais difficile à relire.
- La cinquième erreur consiste à automatiser avant de stabiliser. Un mauvais workflow manuel devient rarement bon parce qu'il est automatisé.

Checklist opérationnelle

Voici une checklist simple pour améliorer immédiatement un dépôt.

- [ ] Créer ou mettre à jour **AGENTS.md**
- [ ] Documenter les commandes de validation
- [ ] Écrire un task brief template
- [ ] Préciser les règles de scope
- [ ] Distinguer les tâches low-risk et high-risk
- [ ] Et demander systématiquement une evidence note après les changements significatifs.

Cette checklist peut être appliquée en une demi-journée sur un projet. Elle ne résout pas tout, mais elle crée une base que l'agent peut lire et que l'équipe peut améliorer.

Conclusion

Un harness de projet rend un agent de codage plus fiable parce qu'il réduit les hypothèses implicites. **AGENTS.md** transforme les conventions d'équipe en contexte actionnable. Les commandes de validation définissent le done de manière exécutable. Les limites de tâche limitent le blast radius. Les permissions alignent le workflow avec le risque réel.

La productivité durable ne vient donc pas d'un prompt parfait. Elle vient d'un dépôt qui sait expliquer comment il doit être modifié, vérifié et relu.

Articles liés

- Un agent de codage n'est pas un chatbot
- Transformer une demande vague en tâche d'ingénierie vérifiable
- Fermer la boucle qualité : tests, review humaine et evidence notes

Références

1. OpenAI Developers, “Custom instructions with AGENTS.md — Codex”, consulté le 2026-06-07.
https://developers.openai.com/codex/guides/agents-md

2. OpenAI Developers, “Custom instructions with AGENTS.md — Codex”, sections sur global guidance et project instructions, consulté le 2026-06-07.
https://developers.openai.com/codex/guides/agents-md

3. OpenAI Developers, “Config basics — Codex”, consulté le 2026-06-07.
https://developers.openai.com/codex/config-basic

4. OpenAI Developers, “Best practices — Codex”, section anti-patterns, consulté le 2026-06-07.
https://developers.openai.com/codex/learn/best-practices