Notilis Tech

Transformer une demande vague en tâche d’ingénierie vérifiable

Transformer une demande vague en tâche d'ingénierie bornée, observable et vérifiable.

2026-05-17 · Task brief · Validation · Qualité

Résumé

La plupart des mauvais résultats avec un agent de codage viennent d’une demande trop floue, trop large ou trop implicite. Si la tâche ne précise pas le contexte, le scope, les contraintes et la validation attendue, l’agent doit deviner.

Cet article propose une méthode pour transformer une intention vague en tâche d’ingénierie vérifiable. Je prends Codex comme cas d’étude, parce que c’est l’agent que j’utilise au quotidien, mais la logique s’applique largement aux autres agents de codage.

Sommaire

Contexte de lecture

Audience

Cet article s’adresse aux développeurs qui transforment des tickets, idées produit ou demandes floues en tâches actionnables pour un agent de codage. Les exemples restent centrés sur Codex comme cas d’usage concret.

Ce que cet article couvre

Il explique comment formuler objectif, contexte, scope, non-goals, contraintes, comportement attendu, chemin de validation et focus de review.

Ce que cet article ne couvre pas

Il ne traite pas de prompting créatif général ni de génération de code hors dépôt réel.

Une demande vague produit un résultat vague

Une demande vague peut produire une réponse convaincante en apparence. C’est même ce qui la rend dangereuse. L’agent peut générer une solution cohérente avec sa propre interprétation, mais pas avec l’intention réelle de l’équipe.

Prenons une demande simple :

```
Améliore le checkout flow.
```

Cette demande peut vouloir dire beaucoup de choses. Améliorer la performance ? Simplifier le code ? Corriger une erreur utilisateur ? Modifier l’UX ? Réduire les appels API ? Renforcer la validation ? Ajouter des tests ? Corriger un bug ? Changer l’ordre des étapes ?

Sans précision, l’agent doit interpréter. Plus il interprète, plus le risque augmente. Avec Codex comme avec d’autres agents de codage, il peut toucher trop de fichiers, modifier un comportement non demandé, changer un contrat API ou produire un refactoring plus large que nécessaire.

Une tâche d’ingénierie exploitable réduit cette interprétation :

```
Corrige le bug de validation du checkout qui permet de soumettre une adresse de facturation
vide après un changement de pays.

Scope:
- Inspecter `frontend/src/checkout`.
- Tu peux modifier les helpers de validation et les tests associés.
- Ne pas changer les payloads de l’API de paiement.
- Ne pas changer l’ordre des étapes du checkout.

Comportement attendu :
- Quand le pays change, la validation de l’adresse de facturation est correctement
réinitialisée.
- Une adresse de facturation vide ne peut pas être soumise.
- Les checkout flows valides existants restent inchangés.

Validation:
- Exécuter les tests de validation du checkout.
- Ajouter ou mettre à jour un test de régression pour ce cas.
- Exécuter le lint frontend.
```

Cette version ne demande pas seulement une amélioration. Elle décrit un comportement attendu, protège les contrats et donne une validation.

Le coût de l’ambiguïté

L’ambiguity cost est le coût créé par ce qui n’est pas dit. Dans un workflow humain, ce coût apparaît sous forme de questions, meetings, retours en review, rework ou dette. Avec un agent de codage, il apparaît souvent sous forme de diff plausible mais difficile à relire.

Le coût de l’ambiguïté augmente avec plusieurs facteurs : taille du scope, criticité du domaine, absence de tests, manque de conventions écrites, dépendances multiples, architecture historique, et faible clarté sur les non-goals.

Une tâche ambiguë n'est pas seulement moins précise. Elle est plus coûteuse à vérifier.

C'est pour cette raison qu'il faut séparer deux moments : la clarification et l'exécution. OpenAI recommande d'ailleurs, lorsqu'on a une idée encore vague, de demander à Codex d'interviewer l'utilisateur, de challenger les hypothèses et de transformer l'idée floue en demande concrète avant d'écrire du code.[1]

Cette étape peut sembler lente, mais elle évite de produire trop tôt. Dans un système logiciel, produire trop tôt n'est pas de la vitesse. C'est souvent du rework différé.

Anatomie d'une bonne tâche d'ingénierie

Une tâche d'ingénierie claire contient plusieurs éléments. Tous ne sont pas nécessaires pour chaque demande, mais plus le risque est élevé, plus la structure doit être explicite.

Élément	Question à laquelle il répond
Objectif	Quel changement veut-on réellement ?
Contexte	Pourquoi ce changement existe-t-il ?
Scope	Où l'agent peut-il lire et modifier ?
Non-goals	Qu'est-ce qui ne doit pas changer ?
Contraintes	Quelles règles doivent être respectées ?
Comportement attendu	Comment le système doit-il se comporter après ?
Chemin de validation	Comment prouver que c'est terminé ?
Focus de review	Où la personne qui relit doit-elle porter son attention ?

L'objectif doit être concret. « Améliorer la maintenabilité » est trop général. « Extraire la logique de formatage des dates dans un helper partagé utilisé par ces trois composants » est vérifiable.

Le contexte doit expliquer le pourquoi sans noyer l'agent. Un bug client, une dette technique localisée, une migration en cours, une convention nouvelle ou une contrainte produit peuvent changer la bonne solution.

Le scope doit borner les fichiers et les modules. Le non-goal doit protéger le reste. Les contraintes doivent préciser les interdictions : pas de dépendance, pas de changement API, pas de migration, pas de refactoring transverse, pas de modification de wording UI si ce n'est pas demandé.

Le comportement attendu doit être observable. Le chemin de validation doit être exécutable autant que possible.

Les non-goals protègent la qualité

Les non-goals sont souvent plus importants que les objectifs. Ils empêchent une tâche locale de devenir un changement transversal.

Un agent qui reçoit « refactor ce service » peut décider de renommer des méthodes, modifier des signatures, extraire des classes, déplacer des fichiers et ajuster les tests. Peut-être que certaines modifications sont bonnes. Mais si l'objectif réel était de supprimer une duplication dans deux méthodes, le résultat devient trop

large.

Les non-goals fixent les frontières :

```
Non-goals:
- Ne pas changer les signatures d’API publiques.
- Ne pas modifier le schéma de base de données.
- Ne pas changer les libellés UI.
- Ne pas refactorer du code non lié.
- Ne pas ajouter de nouvelles dépendances.
- Ne pas modifier la logique d’authentification ou de permissions.
```

Cette section protège la review. Elle permet de dire rapidement si le diff respecte la demande. Elle évite aussi la tentation de « faire mieux » au-delà du besoin.

Dans un projet mature, les non-goals peuvent être standardisés par type de tâche. Par exemple, une tâche de bugfix ne doit pas refactorer massivement. Une tâche de documentation ne doit pas modifier le comportement. Une tâche de test ne doit pas corriger silencieusement la production sans le dire. Une tâche de migration doit lister explicitement les contrats à préserver.

Les critères de done évitent le faux terminé

Le mot « done » est ambigu. Pour un agent de codage, une tâche peut sembler done lorsque le diff est produit. Pour un développeur, elle est done lorsque le comportement est correct, les tests pertinents passent, la review est possible, et les risques connus sont documentés. Pour une équipe, elle peut être done seulement lorsqu’elle est mergée, déployée ou documentée.

Il faut donc préciser les critères de done au niveau de la tâche :

```
Critères de done :
- Le bug est reproduit par un test en échec avant le correctif, ou l’impossibilité de
reproduction est expliquée.
- Le correctif est minimal et limité à la couche de validation.
- Les contrats API existants restent inchangés.
- Les tests pertinents passent.
- La réponse finale inclut les fichiers modifiés, les commandes de validation exécutées et les
limites connues.
```

Ces critères transforment le résultat attendu en checklist de travail. Ils donnent aussi à la personne qui relit un outil de contrôle.

Un bon critère de done ne doit pas être abstrait. « La qualité doit être élevée » ne sert pas beaucoup. « Aucun changement de signature d’API publique » est vérifiable. « Exécuter **npm run test:checkout** » est vérifiable. « Ajouter un test de régression pour l’adresse de facturation vide après changement de pays » est vérifiable.

Lire avant d’écrire

La tentation de demander directement une modification est forte. Pourtant, dans beaucoup de cas, la bonne première étape est de demander à l’agent de lire avant d’écrire.

Lire avant d’écrire ne signifie pas perdre du temps. Cela signifie réduire le risque de casser un pattern existant. Une reconnaissance de codebase peut demander à Codex, ou à un autre agent de codage, d’identifier les fichiers concernés, les patterns existants, les tests disponibles, les commandes probables, les dépendances et

les zones de risque. Ensuite seulement, la modification commence.

Un workflow simple peut être formulé ainsi :

```
Étape 1 : inspecter le code et les tests pertinents.
Étape 2 : résumer le pattern existant.
Étape 3 : proposer un plan de changement minimal.
Étape 4 : attendre confirmation avant d'éditer.
Étape 5 : implémenter le changement approuvé.
Étape 6 : exécuter la validation et produire une evidence note.
```

Ce workflow est particulièrement utile pour les changements complexes, les zones legacy, les modules critiques ou les tâches où le scope est encore incertain.

Pour les tâches simples, il n'est pas toujours nécessaire de bloquer l'exécution. Mais même dans ce cas, on peut demander à l'agent de commencer par inspecter le code et d'expliquer son plan avant d'appliquer le patch.

Demande faible, tâche exploitable

Voici un exemple typique.

Demande faible :

```
Peux-tu nettoyer ce composant ?
```

Task exploitable :

```
Objectif :
Réduire la duplication dans `UserProfileCard` sans changer le comportement.

Contexte :
Le composant duplique la logique d'affichage du statut utilisateur et du type de compte. On
veut un petit refactoring avant d'ajouter un nouveau statut plus tard.

Scope:
- `frontend/src/components/UserProfileCard.tsx`
- Tests unitaires associés uniquement.

Non-goals:
- Ne pas changer le layout visuel.
- Ne pas changer les libellés.
- Ne pas changer les types API.
- Ne pas ajouter de dépendances.

Comportement attendu :
Le composant rend le même résultat pour tous les cas de test existants.

Validation:
- Exécuter `npm test -- UserProfileCard`.
- Exécuter `npm run lint`.
- Signaler toute commande de validation qui ne peut pas être exécutée.

Focus de review :
Vérifier que le comportement reste inchangé et que le helper extrait ne rend pas le composant
plus difficile à lire.
```

Cette version n’est pas seulement plus longue. Elle est plus contrôlable. Elle indique à l’agent le résultat, les limites et la preuve attendue.

Template de task brief réutilisable

Un template standard évite de réinventer la structure à chaque demande :

```
# Task brief d’ingénierie

## Objectif

Que doit-on changer ?

## Contexte

Pourquoi ce changement est-il nécessaire ?

## Scope

Quels fichiers, dossiers ou modules sont dans le scope ?

## Non-goals

Qu’est-ce qui ne doit pas changer ?

## Contraintes

Contraintes d’architecture, de dépendances, de contrats API, de sécurité, de performance ou de compatibilité.

## Comportement attendu

Quel comportement observable doit exister après le changement ?

## Validation

Commandes, tests, reproducteur, build, lint ou smoke checks à exécuter.

## Focus de review

Que doit inspecter attentivement la personne qui relit ?

## Evidence attendue

Que doit rapporter l’agent à la fin ?
```

Il est préférable de garder ce template dans le dépôt, dans un dossier **docs/workflows/**, **.codex/**, ou tout autre emplacement utilisé par l’équipe. L’important est qu’il soit versionné et facile à copier.

Quand demander à l’agent de clarifier

Toutes les tâches ne doivent pas être entièrement définies avant de démarrer. Il est parfois utile de demander à Codex, ou à l’agent utilisé par l’équipe, de clarifier. Mais cette clarification doit être explicite.

Une bonne demande peut être :

```
Avant d'écrire du code reformule ma demande pour que je sois certain qu'on est alignés. Après ta reformulation,pose jusqu'à cinq questions si le scope, le comportement attendu ou le chemin de validation est ambigu.
Si la tâche est assez claire, propose un plan court puis avance.
```

Ou, pour une tâche plus sensible :

```
Ne modifie pas encore les fichiers.
Commence par reformuler ma demande telle que tu l'as comprends.
Inspecte ensuite le dépôt et produis :
- fichiers probablement concernés
- patterns existants
- risques
- informations manquantes
- chemin de validation proposé
```

Cette approche transforme Codex en assistant de cadrage avant de devenir agent d'exécution. Le principe vaut aussi pour les autres agents de codage : avant de produire un diff, ils doivent parfois aider à rendre la tâche vérifiable. C'est particulièrement utile lorsque le ticket initial vient d'un non-développeur, d'un backlog produit ou d'un incident mal documenté.

Les erreurs fréquentes

- La première erreur consiste à confondre intention et tâche. « Rendre ça plus robuste » est une intention. « Ajouter un test de régression pour ce cas et corriger la validation sans changer le payload API » est une tâche.
- La deuxième erreur consiste à oublier les non-goals. Si l'on ne dit pas ce qui ne doit pas changer, il devient difficile de reprocher à l'agent d'avoir élargi le scope.
- La troisième erreur consiste à demander une grande feature en une seule passe. Mieux vaut découper : reconnaissance, plan, première slice, validation, review, slice suivante.
- La quatrième erreur consiste à ne pas fournir de chemin de validation. Une tâche sans validation produit un diff, pas une preuve.
- La cinquième erreur consiste à laisser les décisions dans le chat. Les décisions importantes doivent finir dans le task brief, l'evidence note ou la PR.

Checklist opérationnelle

Avant de lancer une tâche avec un agent de codage, il faut vérifier que

- ☐ l'objectif est concret
- ☐ le scope est limité
- ☐ les non-goals sont explicites
- ☐ les contraintes sont écrites
- ☐ le comportement attendu est observable
- ☐ la validation est connue

- [ ] la personne qui relit sait quoi inspecter

Si l’un de ces éléments manque, il vaut mieux demander à l’agent de clarifier ou de proposer un plan, plutôt que de le laisser modifier le code immédiatement.

Conclusion

Une bonne demande à un agent de codage n’est pas un prompt plus élégant. C’est une tâche d’ingénierie vérifiable.

Le développeur garde la responsabilité de transformer l’intention en scope, contraintes, non-goals, comportement attendu et chemin de validation. Cette discipline réduit l’ambiguity cost, limite le blast radius, accélère la review et rend le travail de l’agent plus fiable.

Le bon réflexe est donc simple : ne pas demander à Codex, ni à aucun autre agent, de deviner ce que l’équipe sait déjà. Écrire la tâche de façon à ce que le résultat puisse être produit, vérifié et relu.

Articles liés

- Un agent de codage n'est pas un chatbot
- Construire un harness de projet : **AGENTS.md**, commandes de validation et limites de tâche
- Fermer la boucle qualité : tests, review humaine et evidence notes

Références

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