WEB & $EO

EBOOK · 2026

GUIDE PRATIQUE · GRATUIT

# FAIRE VIVRE TON PROJET VIBE CODING EN 6 ÉTAPES, DU TICKET À LA PRODUCTION

Le process complet pour faire évoluer ton app dans le temps

PAR

Maximilien Labadie

webandseo.fr

# FAIRE VIVRE TON PROJET VIBE CODING EN 6 ÉTAPES



ÉTAPE 1

## Cadrer l'après-production : pourquoi le vrai problème commence après le lancement

Avec le vibe coding, créer une application et la mettre en production est devenu facile. Le vrai problème commence juste après : ton application vit, et il va falloir la faire évoluer. Nouvelles features demandées par tes clients, bugs à corriger, retours utilisateurs qui s'accumulent : c'est là que la majorité des projets vibe coding partent à la catastrophe. Le scénario classique : tu as une idée de nouvelle feature, tu demandes à ton agent IA de la développer, et tu la pousses directement en production sans phase de test. Résultat garanti : du code non testé en prod, des régressions sur des features qui marchaient, et un projet qui devient impossible à maintenir dans le temps. Un projet vivant suit un cycle précis : une phase de recette après le lancement, puis une phase de run où l'application tourne, puis les versions suivantes (V2, V3, V4). Chaque évolution doit traverser les mêmes étapes : collecte du besoin sous forme de ticket, regroupement en lot PRD, développement par l'IA, test en local, recette client en staging, et seulement ensuite déploiement en production. Le signal que ton process est sain : aucune modification n'arrive en production sans être passée par au moins un environnement intermédiaire et une validation humaine. Le piège à éviter absolument : demander à l'IA de modifier le code et de déployer directement. Même si elle te dit que c'est fait et testé, tu perds toute traçabilité et toute capacité de recette. L'objectif n'est pas d'aller vite une fois, c'est d'avoir quelque chose de maintenable pendant des années.

*→ Installe un système de tickets dès le jour de la mise en production, pas quand les retours deviennent ingérables. Règle simple : chaque bug ou idée de feature devient un ticket rattaché à un environnement (local, staging ou production) avant toute demande à l'IA. Attends d'avoir au moins 2 ou 3 tickets sur un même périmètre avant de créer un lot PRD : tu évites de lancer l'agent pour des micro-modifications isolées, et tu obtiens un PRD plus cohérent qui traite un vrai sujet plutôt qu'une rustine.*

ÉTAPE 2

## Mettre en place ton outil de ticketing inspiré de Userback

Ton premier chantier, c'est de te doter d'un outil de ticketing adapté au vibe coding. Le modèle de référence, c'est Userback : un outil pensé pour les SaaS et les applications, qui permet aux clients de remonter des retours sur des features ou des bugs, très utile quand plusieurs personnes travaillent sur un projet, notamment avec quelqu'un chargé de la recette. Reprends ce modèle et transpose-le à ta façon de créer : un tableau de bord avec un système de tickets organisé en Kanban, personnalisable selon les projets.

Ton outil doit contenir quatre briques indispensables :
- un Kanban par projet, avec au minimum une colonne « Reçu » où arrivent les nouveaux tickets, puis des statuts qui avancent tout seuls : « PRD », « En cours », « Livré » ;
- une gestion multi-projets, pour suivre plusieurs clients ou applications depuis le même tableau de bord ;
- des lots PRD : tu sélectionnes plusieurs tickets et tu les regroupes en un seul lot, qui deviendra le document de spécifications transmis à ton agent IA ;
- des changelogs, générés à partir des PRD traités, pour alerter tes clients que leurs tickets ont été résolus, avec un résumé de ce qui a été fait.

Ne cherche pas la sophistication au départ : la solution peut rester basique, l'important c'est le process qu'elle structure. Tu sais que c'est bon quand tu peux créer un projet, voir arriver un ticket dans « Reçu », le regrouper dans un lot PRD et suivre son statut sans toucher toi-même au Kanban. Piège classique : vouloir gérer l'évolution d'une app en production par messages ou par mémoire, sans tickets formalisés, ce qui mène droit à la catastrophe dès que les demandes s'accumulent.

*→ Commence par un seul projet pilote dans ton outil (par exemple l'app que tu viens de mettre en production) et fais-y transiter au moins 2 tickets réels de bout en bout avant d'y brancher tes autres clients. Objectif : valider que le cycle complet « ticket reçu, lot PRD, livré, changelog » fonctionne. Ne migre le reste de tes projets qu'une fois cette boucle rôdée.*

ÉTAPE 3

## Structurer tes projets et tes versions (recette, run, V2, V3...)

Le point de départ : une fois ton app en production, ne mélange plus jamais tout dans un seul espace. Ton projet doit être découpé en phases et en environnements clairs, sinon tu pars droit vers la catastrophe dès la première évolution.

Commence par créer un projet par application dans ton outil de suivi. Dans Ticket, chaque client ou chaque app a son propre projet, avec son Kanban personnalisable : c'est lui qui fait vivre les tickets du dépôt jusqu'à la livraison.

Ensuite, structure le cycle de vie de ton app en trois temps :
- La recette : la phase où une feature fraîchement livrée est testée avant d'aller en production. C'est ici que le client (ou toi) remonte les problèmes.
- Le run : l'application vit en production. Les tickets qui arrivent sont des bugs ou des améliorations du quotidien.
- Les versions suivantes : V2, V3, V4. Chaque grosse évolution mérite son propre espace de travail.

Puis crée tes environnements, c'est ultra important :
- Le local / dev : ta machine, réservé à toi. Le client n'y a pas accès. Si tu vois des choses à modifier sans avoir le temps de lancer un agent dessus, tu remontes un ticket, et une fois que tu en as suffisamment, tu crées ton lot PRD.
- Le staging : entre le local et la production, dédié au client. Tu y livres une feature, il la teste en recette, il remonte des tickets, et tu sais immédiatement que ces tickets viennent de la recette sur cette feature précise.
- La production : la version live, celle que tout le monde utilise.

Règle d'or : chaque ticket est rattaché à un environnement. Comme le code et les features peuvent différer entre dev, staging et production, tu dois toujours savoir d'où vient un ticket pour savoir comment le traiter.

Variante avancée : au lieu d'environnements par stade, crée un environnement par version. La production correspond au run, et la V2 qui arrive a son propre environnement, complètement indépendant. Ça suit la logique des branches sur GitHub, et tu peux avoir plusieurs environnements dans le même projet sans jamais qu'ils se parasitent.

Tu sais que ta structure est bonne quand tu peux répondre instantanément à deux questions : « ce bug a été remonté où ? » et « cette feature en est à quelle phase ? ». Le piège classique : tout faire vivre dans un seul environnement et pousser directement en prod sans recette. Résultat garanti : des régressions en live et aucun moyen de revenir proprement en arrière.

*→ Adapte le nombre d'environnements à la taille de ton projet : en solo sur une petite app, 2 environnements suffisent (local + production) ; dès qu'un client ou un utilisateur teste pour toi, ajoute systématiquement un staging entre les deux. Et si tu prépares une V2 pendant que la V1 tourne, crée un environnement dédié par version, aligné sur tes branches GitHub : chaque environnement reste indépendant, et un bug remonté en V2 ne pollue jamais le suivi de la production.*

ÉTAPE 4

## Configurer la gestion multi-environnements (local, staging, production)

Le point de bascule entre un projet vibe coding qui vit et un projet qui explose, c'est la séparation des environnements. Tu configures trois espaces indépendants au sein d'un même projet : le local, le staging et la production.

Le local (ou dev) : c'est ton terrain de jeu, sur ta machine, invisible pour le client. C'est là que tu testes les modifications livrées par l'IA avant toute exposition. Quand tu repères toi-même des choses à modifier sans avoir le temps de lancer un agent tout de suite, tu remontes un ticket rattaché à l'environnement local, et tu accumules jusqu'à avoir de quoi constituer un lot PRD.

Le staging : c'est l'espace intermédiaire, dédié à la recette client. Quand tu livres une feature, le client la teste là-dessus et remonte ses tickets depuis cet environnement. Tu sais immédiatement que ces retours concernent la phase de recette de la feature en cours, et non un bug en production.

La production : c'est la version live, celle que les utilisateurs finaux utilisent. Rien n'y arrive sans être passé par le local puis validé en staging.

Chaque ticket est obligatoirement rattaché à un environnement. Comme le code et les features diffèrent entre dev, staging et prod, tu dois toujours savoir d'où vient le retour pour savoir comment le traiter. C'est la règle non négociable.

Variante possible : un environnement par version. Tu gardes la production en run, et tu crées un environnement dédié à la V2 en préparation. Ça suit la logique des branches GitHub, chaque environnement restant complètement indépendant dans le même projet.

Tu sais que ta configuration est bonne quand : chaque ticket remonté affiche clairement son environnement d'origine, ton client n'a accès qu'au staging pour la recette, et aucune modification de l'IA ne peut atterrir en production sans passer par une pull request testée en local puis validée en staging.

Pièges courants : laisser l'IA pousser directement en prod (aucun test, catastrophe garantie), mélanger les retours de recette et les bugs de prod dans la même file, et oublier que ton agent peut occuper un port (le port 3000 déjà pris au lancement de `npm run dev`, c'est souvent lui qui a démarré le serveur avant toi).

*→ Bloque la promotion en amont : configure ton dépôt GitHub pour que la branche de production n'accepte que des merges venant du staging, lui-même alimenté uniquement par des pull requests en brouillon relues par toi. Concrètement : 1 PR par lot PRD, review humaine obligatoire avant merge, puis promotion dev vers staging, staging vers prod, chaque étape étant un merge séparé et traçable dans l'historique des commits.*

ÉTAPE 5

## Installer le widget de collecte sur ton site

Le widget est la porte d'entrée de tout ton process : c'est lui qui permet de remonter un bug ou une idée d'amélioration directement depuis ton site, sans quitter la page, et de faire atterrir cette information automatiquement dans ton outil de ticketing, dans la colonne « Reçu ». Sans lui, tu retombes dans le chaos classique : des captures d'écran envoyées par mail, des messages perdus dans une conversation, aucun contexte technique.

Concrètement, tu installes le widget sur chaque environnement de ton projet, car le point clé du système, c'est le multi-environnements : chaque ticket est rattaché à l'environnement depuis lequel il a été remonté.

- le local / dev : réservé à toi, sur ta machine. Quand tu vois quelque chose à modifier et que tu n'as pas le temps de lancer un agent dessus, tu remontes un ticket via le widget. Une fois que tu en as accumulé suffisamment, tu crées ton lot PRD.
- le staging : l'espace de recette pour ton client. Tu livres une feature, il la teste, il remonte ses tickets, et tu sais immédiatement qu'ils viennent de la phase de recette sur la feature que tu viens de créer.
- la production : l'environnement vivant, celui où ton app tourne réellement.

Variante utile : crée un environnement par version (production pour le run, puis un environnement V2, V3...). Ça suit la logique des branches sur GitHub, et chaque environnement reste complètement indépendant au sein du même projet.

Pour remonter un ticket, la manipulation est volontairement simple : le widget récupère automatiquement les informations de ton compte, tu entoures la zone concernée à l'écran (tu peux en ajouter plusieurs, mais une seule suffit souvent), tu tapes une demande courte et explicite du type « Améliorer le design du tableau », et tu envoies. Côté ticket, tu retrouves tout le contexte : l'environnement, la zone sélectionnée, les logs capturés qui serviront à l'IA, plus un espace de commentaires pour les échanges avec le client.

Tu sais que c'est bon quand : le ticket arrive dans la colonne « Reçu » du bon environnement, avec la capture de zone et les logs attachés. Piège classique à éviter : multiplier les gros tickets fourre-tout. Prends des petits éléments, un problème par ticket, ça rend la génération du PRD bien plus propre derrière.

*→ Combien de tickets avant d'agir ? Ne traite jamais un ticket isolé : laisse-les s'accumuler dans « Reçu » et regroupe-les par zone ou par thème (par exemple 2 à 5 tickets sur l'interface du studio) pour créer un seul lot PRD. L'agent génère alors un PRD structuré en clusters, avec chaque cluster rattaché à ses tickets, ce qui évite de lancer une génération complète pour une simple retouche de design. Et formule tes demandes en une phrase actionnable (« Améliorer le design du tableau » plutôt que « C'est moche ») : c'est ce texte que l'IA analysera pour rédiger le PRD.*

ÉTAPE 6

## Remonter un ticket : zone entourée, description, logs et environnement

Le point de départ de tout le process, c'est le ticket. Sans ticket bien remonté, l'IA ne peut rien faire de propre derrière. Concrètement, tu installes sur ton site un petit widget rattaché à ton outil de ticketing : dès qu'une information est remontée depuis le site, elle arrive automatiquement dans la colonne « Reçu » de ton tableau Kanban. Voici comment remplir un ticket exploitable.

1. Entoure la zone concernée. Le widget te permet de sélectionner visuellement une ou plusieurs zones de l'écran : un tableau moche, un bouton mal placé, un bloc à revoir. Tu peux ajouter plusieurs zones sur un même ticket si le problème en touche plusieurs. Pour un retour simple, une seule zone suffit.
2. Rédige une description courte et orientée résultat. Pas besoin d'un roman : « améliorer le design du tableau » ou « améliore le tableau pour qu'il soit plus lisible » suffisent, parce que le reste du contexte est capturé automatiquement. L'erreur classique, c'est de décrire la solution technique au lieu du problème visible : décris ce que tu vois et ce que tu attends, l'agent se chargera du « comment ».
3. Laisse les logs se capturer tout seuls. Quand tu ouvres le ticket dans l'outil, tu retrouves tout ce qui a été saisi : la zone sélectionnée, la description, mais aussi les logs capturés au moment de la remontée. Ce sont ces logs qui serviront à l'agent pour comprendre le contexte technique sans que tu aies à copier-coller quoi que ce soit.
4. Vérifie l'environnement rattaché. C'est le point ultra important : chaque ticket est rattaché à un environnement (local/dev, staging ou production), parce que le code et les features peuvent différer d'un environnement à l'autre. Si tu remontes un bug depuis la prod, le ticket arrive tagué « production » ; s'il vient de la phase de recette client, il arrive tagué « staging ». Tu sais ainsi immédiatement d'où vient le problème et comment le traiter. Tu peux aussi créer un environnement par version (une V2 en préparation, par exemple), chaque environnement restant complètement indépendant dans le même projet.
5. Utilise les commentaires si besoin. Si le client n'a pas compris quelque chose ou te réexplique sa demande, ces échanges sont conservés sur le ticket et seront pris en compte lors de la génération du PRD.

Tu sais que ton ticket est bon quand, en l'ouvrant, tu retrouves sans effort : la zone entourée, une description compréhensible par quelqu'un qui n'a pas vu l'écran, les logs, et le bon environnement. Piège courant : accumuler les tickets vagues du type « ça marche pas » sans zone ni contexte, ce qui oblige l'agent à deviner et dégrade la qualité du PRD généré ensuite.

*→ Combien de tickets avant d'agir : ne lance pas un agent pour chaque ticket isolé. Laisse les tickets s'accumuler dans la colonne « Reçu » (sur le local, remonte-toi tes propres tickets via le widget dès que tu vois un truc à modifier sans avoir le temps de le traiter), puis sélectionne-en plusieurs et crée un lot PRD unique. Deux tickets proches, comme deux améliorations de tableaux, se traitent très bien dans le même lot : tu divises le nombre de cycles de génération, de livraison et de pull requests, donc le temps passé.*

ÉTAPE 7

## Regrouper les tickets en lot PRD

Une fois que tes tickets remontent dans la colonne « Reçu » de ton outil de ticketing, ne les traite jamais un par un à la volée : regroupe-les en ce que j'appelle un lot PRD. Concrètement, tu sélectionnes plusieurs tickets liés (par exemple deux demandes d'amélioration de design sur une même page) et tu les rattaches à un seul lot, qui deviendra le document de spécification que tu donnes à ton agent IA.

Voici le process, étape par étape :
- Filtre tes tickets par environnement : un lot PRD doit concerner un seul environnement (production, staging ou local), sinon tu mélanges des demandes de recette client avec des bugs de prod, et l'agent ne sait plus quoi prioriser.
- Sélectionne les tickets cohérents entre eux : dans mon exemple, j'avais remonté deux tickets depuis le widget (« Améliorer le design du tableau » sur la création d'article et sur les applications script), je les ai cochés tous les deux et j'ai cliqué sur « créer un lot PRD ».
- Choisis entre créer un nouveau lot ou rattacher à un lot existant : si un lot sur le même sujet est déjà ouvert, ajoute le ticket dedans plutôt que d'en créer un doublon.
- Vérifie le contenu avant génération : chaque ticket apporte sa zone entourée à l'écran, sa description, ses logs capturés automatiquement et les éventuels commentaires échangés avec le client. Tout ça sera analysé par l'agent pour rédiger le PRD.

Une fois le lot créé, tu rentres dedans et tu as un aperçu des demandes qu'il contient. C'est volontairement basique à ce stade : l'agent n'a encore rien fait. Le signal de réussite, c'est que chaque ticket du lot est bien rattaché et que tu peux cliquer sur chacun pour revoir la zone, la description et les logs. Piège classique : créer un lot avec un seul ticket trivial. Un lot PRD a du sens à partir de 2 ou 3 demandes regroupées ; en dessous, tu multiplies les cycles de génération, de review et de pull request pour rien.

*→ Attends d'avoir au minimum 2 ou 3 tickets cohérents sur un même environnement avant de créer ton lot PRD. Dans mon exemple, j'ai remonté deux tickets design depuis le widget en production et je les ai regroupés en un seul lot, ce qui a généré une seule pull request au lieu de deux cycles complets. Et regroupe toujours par environnement : un ticket remonté en staging pendant la recette ne va jamais dans le même lot qu'un bug de production.*

ÉTAPE 8

## Générer le PRD automatiquement (annotations, commentaires, clusters)

Une fois ton lot PRD créé à partir des tickets sélectionnés, tu lances la génération automatique du PRD. Un agent démarre et fait le travail d'analyse à ta place : il récupère les transcriptions (utile si le ticket contient une vidéo de démonstration), il analyse les zones que tu as entourées dans le widget, les descriptions saisies et tous les commentaires échangés sur le ticket. Ce dernier point est crucial : si le client n'a pas compris quelque chose et te réexplique en commentaire, ou si toi-même tu précises un point, ces échanges sont pris en compte dans le run de génération. Plus le ticket est riche en contexte, plus le PRD est qualitatif.

Le processus est transparent grâce à un système de logs : tu vois en temps réel où en est l'agent, quelle phase il exécute (analyse des captures, analyse des échanges, rédaction). La génération prend un peu de temps, c'est normal : l'IA analyse chaque élément puis rédige un document structuré. Ne l'interromps pas, même si ça semble long sur des tickets simples.

Le résultat est organisé en clusters : chaque cluster regroupe une demande cohérente (par exemple « amélioration UX du tableau Studio App ») et est rattaché automatiquement au ticket correspondant. Tu retrouves les annotations remontées, tout est expliqué et structuré. Pour chaque ticket, le PRD généré contient : le contexte, ce qui a été observé à l'écran, l'étape de reproduction, la cause possible, la correction attendue, et surtout le fichier, le sélecteur et le composant concernés. C'est ce niveau de précision technique qui permet à l'agent de développement de travailler sans aller-retour.

Enfin, le PRD est livré en plusieurs formats complémentaires : une version aperçu pour une lecture rapide dans l'outil, une version Markdown optimisée pour être lue par l'IA, une version PDF destinée au client, et la préparation d'une pull request côté GitHub pour la suite du process. Tu sais que c'est bon quand chaque cluster est rattaché à son ticket et que chaque demande mentionne un fichier et un composant précis. Piège classique : lancer la génération sur un ticket vide ou sans zone annotée, tu obtiendras un PRD vague que l'agent traitera mal.

*→ Avant de lancer la génération, ajoute systématiquement un commentaire explicite sur chaque ticket : contexte, comportement attendu, comportement actuel. Exemple : « Le tableau s'affiche, mais le design est brut. Attendu : en-tête en capitales, style cohérent avec le reste du studio. » Ces échanges sont injectés dans le run de génération et font la différence entre un PRD générique et un PRD qui cite le fichier et le sélecteur exacts à modifier. Vérifie ensuite dans l'aperçu que chaque cluster contient bien les champs fichier, sélecteur et composant : si l'un manque, enrichis le ticket et régénère.*

ÉTAPE 9

## Vérifier les livrables du PRD (aperçu, Markdown, PDF, PR)

Une fois la génération terminée, ton agent produit quatre livrables distincts, et chacun a un rôle précis dans ton process. Ne les survole pas : c'est à ce moment-là que tu détectes un PRD mal construit avant de gaspiller un cycle de développement. 1) L'aperçu : c'est ta version de lecture rapide, en interne. Vérifie que le PRD est bien découpé en clusters (par exemple « Amélioration UX du tableau Studio App ») et que chaque cluster est rattaché aux tickets d'origine. Tu sais que c'est bon quand chaque ticket retrouve son cluster et que rien n'est orphelin. 2) Le Markdown : c'est la version destinée à l'IA, celle que ton agent de développement va lire en local. Contrôle sa structure : contexte, ce qui a été observé à l'écran, étapes de reproduction, cause possible, correction attendue, et surtout les fichiers, sélecteurs et composants concernés. Si ces éléments techniques manquent, l'agent codera à l'aveugle. 3) Le PDF : c'est la version client, pour qu'il suive ce qui va être fait sans jargon. 4) La pull request : elle sera créée automatiquement sur GitHub après traitement, en mode brouillon (draft), et reprend ce qui change, ce qui a été construit et la note associée. Avant d'approuver, fais défiler chaque cluster et vérifie que les annotations remontées par l'agent correspondent bien à tes demandes initiales. Piège classique : approuver un PRD qui mélange deux sujets sans rapport dans le même cluster, ce qui donnera une PR fourre-tout impossible à recetter. Autre signal d'alerte : un cluster sans ticket rattaché, signe que l'agent a extrapolé une demande qui n'existe pas. Quand les quatre livrables sont cohérents et complets, tu peux approuver le PRD et lancer la livraison sereinement.

*→ Compte les clusters avant d'approuver : si tu as remonté 2 tickets, tu dois avoir 1 à 2 clusters maximum, chacun rattaché à au moins un ticket. Au-delà, l'agent a découpé n'importe comment ou halluciné des demandes : régénère le PRD plutôt que de l'approuver. Cette vérification prend 2 minutes et t'évite une pull request inutilisable.*

---

ÉTAPE 10

## Livrer le PRD et le récupérer en local

Une fois le PRD généré et validé, tu l'approuves dans ton outil de ticketing et tu lances la livraison. Le PRD n'est pas envoyé par mail ou copié-collé : il est déposé dans un dossier dédié de ton dépôt Git. Pour le récupérer en local, tu ouvres ton projet sur ta machine et tu fais un simple `git pull` pour récupérer la dernière version : le dossier « PRD » se met à jour automatiquement, avec par exemple un fichier « PRD Prod 7 » qui contient toutes les informations transmises (contexte, zones annotées, descriptions, commentaires, fichiers et composants concernés). Ensuite, tu lances ton assistant IA en local (un plugin ou une commande dédiée, par exemple « ticket dev ») avec pour mission de lire le PRD, traiter les demandes une par une, puis mettre à jour le statut dans l'outil de ticketing. Dès que tu rafraîchis l'interface, le lot passe de « PRD » à « En cours », puis à « Livré » quand le travail est terminé : la synchronisation est automatique, tu n'as rien à déplacer à la main. Point crucial : l'IA ne pousse jamais directement en production. Elle travaille en local, teste ce qu'elle peut, puis crée une pull request en brouillon sur GitHub, sans conflit, qui reprend ce qui change et ce qui a été construit. Tu sais que c'est bon quand le statut est passé en « Livré » dans ton outil, que la PR apparaît dans l'onglet dédié, et que tu peux lancer `npm run dev` pour vérifier le résultat sur ta machine. Piège classique : si le port 3000 est déjà occupé au lancement du serveur, c'est souvent que l'agent l'a déjà démarré lui-même, vérifie avant de tuer le process. Autre piège : si l'agent te répond en anglais ou retombe sur un outil de test par défaut (par exemple une extension de navigateur alors que tu utilises un autre outil de navigation), précise-lui explicitement la langue et l'outil à utiliser dans tes instructions.

*→ Structure ta commande locale en trois instructions explicites : « Lis le fichier PRD présent dans le dossier /PRD, implémente chaque demande, teste le rendu dans le navigateur, puis crée une pull request en brouillon sans jamais merger ni pousser en production ». Ajoute « réponds-moi en français » dès le premier message si l'agent démarre en anglais : ça t'évite de relire des comptes rendus dans une langue qui te ralentit, et la consigne reste active pour toute la session.*

---

ÉTAPE 11

## Lancer l'agent de développement et suivre les statuts automatiques

Une fois le PRD livré, l'IA doit pouvoir le récupérer toute seule : c'est là que tout se joue. Concrètement, tu commences par te placer sur ton projet en local et tu fais un pull pour récupérer la dernière version du code. Ton dossier « PRD » se met alors à jour automatiquement : le nouveau PRD (par exemple « PRD Prod 7 ») vient de s'y ajouter, avec toutes les informations transmises (descriptions, zones annotées, commentaires échangés avec le client). L'IA a accès à tout, tu n'as rien à copier-coller.

Ensuite, tu lances ton agent de développement (dans le process, c'est un plugin appelé « ticket dev » rattaché à l'outil de ticketing). Son travail se déroule en 4 temps :

- il lit le PRD en intégralité : contexte, élément observé à l'écran, étape de reproduction, cause possible, correction attendue, fichier, sélecteur et composant concernés ;
- il fait les modifications dans le code ;
- il teste ce qu'il a fait ;
- il met à jour le statut du lot dans l'outil, au fur et à mesure de sa progression.

Les statuts changent automatiquement : le lot passe de « PRD » à « En cours » dès que l'agent démarre (un simple rafraîchissement de la page suffit pour le vérifier), puis à « Livré » quand il a terminé. Tu n'as jamais à déplacer quoi que ce soit manuellement : le tableau Kanban n'est qu'une vue, tout se met à jour en arrière-plan.

Point crucial : l'agent ne pousse jamais directement en production. À la place, il crée une pull request en brouillon sur GitHub, qui reprend ce qui change, ce qu'il a construit et ses notes. Tu peux vérifier dans l'onglet « PR » de ton outil que la PR est bien créée et sans conflit. Tu sais que c'est bon quand : le statut affiche « Livré », la pull request existe sur GitHub en draft, et le fichier d'index de ton projet a été mis à jour par l'agent pour indiquer que le traitement est terminé.

Pièges courants : l'agent peut échouer à tester s'il s'appuie sur une extension de navigateur qui n'est pas ouverte (préfère un outil de navigation dédié type Dev Browser plutôt que l'intégration native de Chrome) ; il peut aussi lancer lui-même le serveur de développement, ce qui bloque ton `npm run dev` avec un port 3000 déjà occupé : pense à vérifier avant de le relancer. Enfin, n'hésite pas à dialoguer avec lui pendant le run : s'il te répond en anglais, demande-lui simplement de continuer en français.

*→ Ne touche à rien pendant que l'agent tourne : ton seul réflexe est de rafraîchir la page de l'outil toutes les 2-3 minutes pour suivre la progression des statuts (« En cours » puis « Livré »). Le signal de fin de run à vérifier systématiquement : la pull request en brouillon sur GitHub + le statut « Livré » + l'absence de conflit signalée dans la PR. Si l'un des trois manque, c'est que l'agent a calé en route : rouvre le dialogue et demande-lui où il en est avant de relancer quoi que ce soit.*

ÉTAPE 12

## Dialoguer avec l'agent et vérifier la qualité du plan de correction

Une fois le plugin « ticket dev » lancé, l'agent lit le PRD et te propose un plan de correction avant de toucher au code. Ton rôle ici n'est pas de coder, mais de contrôler : tu es le chef de projet qui valide le plan avant exécution. Concrètement, un plan de qualité doit contenir, pour chaque ticket, les éléments suivants : le contexte, ce qui a été observé à l'écran, les étapes de reproduction, la cause possible, la correction attendue, et surtout le fichier, le sélecteur ou le composant concerné. Si l'un de ces blocs manque, renvoie l'agent au travail : un plan vague produit toujours une correction vague.

Pendant le dialogue, n'hésite pas à recadrer l'agent immédiatement sur la forme : s'il te répond en anglais alors que ton projet est en français, dis-le-lui directement (« réponds-moi en français ») plutôt que de te fatiguer à traduire. Chaque échange est pris en compte dans le run, donc tes précisions améliorent le résultat final. Tu peux valider un plan partiellement : « le point 1 est bon, vas-y, mais revois le point 2, tu n'as pas ciblé le bon composant ».

Tu sais que le plan est bon quand : chaque ticket du lot est rattaché à un cluster identifié, chaque correction mentionne un fichier ou un sélecteur précis (pas juste « améliorer le design »), et les étapes de reproduction correspondent à ce que tu as réellement constaté. Piège classique : l'agent peut échouer à tester visuellement s'il retombe sur un mauvais outil de navigateur (par exemple une extension Chrome alors que tu utilises un autre navigateur de dev). Vérifie dans ses retours qu'il a bien pu ouvrir la page et tester, sinon considère la correction comme non vérifiée. Signal de fin d'étape : l'agent modifie l'`index.json` de ton outil de ticketing pour passer le statut à « Livré » et crée la pull request en brouillon sur GitHub.

*→ Avant de valider le plan, exige systématiquement 6 blocs par ticket : contexte, observation, reproduction, cause possible, correction attendue, fichier/composant ciblé. Compte-les : si tu n'en trouves que 4 ou 5, renvoie un message du type « ton plan est incomplet, il manque la cause possible et le fichier ciblé pour le ticket 2, refais-le ». Et dès le premier échange, impose la langue : « toutes tes réponses en français », ça t'évite de lire des pavés en anglais à chaque itération.*

ÉTAPE 13

## Contrôler la pull request créée automatiquement sur GitHub

Une fois que ton agent a terminé son travail sur le PRD, il ne pousse jamais directement en production : il crée une pull request sur GitHub. C'est un point de contrôle fondamental, et ton rôle à ce stade est de la vérifier avant toute chose.

Voici le déroulé précis :
1. Ouvre ton dépôt GitHub (par exemple celui de ton projet) et va dans l'onglet « Pull requests ». La PR créée par l'agent peut mettre quelques secondes à apparaître : rafraîchis la page si tu ne la vois pas tout de suite.
2. Vérifie que la PR correspond bien au PRD traité : son titre et sa description reprennent ce qui change, ce que l'agent a construit, et ses notes de travail.
3. Contrôle qu'elle est en statut « draft » (brouillon). C'est volontaire : tant qu'elle est en brouillon, personne ne peut la merger par accident. C'est ta barrière de sécurité.
4. Vérifie l'absence de conflits : GitHub l'indique clairement sous la PR. S'il n'y a pas de conflit avec la branche cible, tu peux avancer sereinement.
5. Dans ton outil de ticketing, onglet « PR », tu dois retrouver cette même pull request en brouillon, et le lot PRD doit être passé automatiquement au statut « Livré ». C'est le signal que la synchronisation entre GitHub et ton système fonctionne.

Tu sais que c'est bon quand : la PR est en draft, sans conflit, sa description reflète fidèlement les tickets du lot, et le statut « Livré » est remonté tout seul dans ton tableau.

Pièges courants : merger la PR sans l'avoir testée en local d'abord (lance ton serveur avec `npm run dev` et vérifie visuellement chaque modification, car l'agent peut avoir touché une zone différente de celle que tu attendais, par exemple l'intérieur d'un tableau plutôt que son en-tête). Autre piège : le port 3000 déjà occupé parce que l'agent a lui-même lancé un serveur pendant son run ; tue le processus existant ou change de port avant de conclure que « ça ne marche pas ». Enfin, si le rendu visuel est décevant, ne merge pas « pour voir » : remonte un ticket de correction, tu restes dans le même PRD et tu repars sur un cycle propre.

*→ Ne clique jamais sur « Merge » tant que tu n'as pas lancé `npm run dev` et inspecté chaque zone modifiée dans le navigateur, une par une, en comparant avec la description de la PR. Prévois 5 à 10 minutes de recette visuelle par ticket contenu dans le lot. Et si le serveur refuse de démarrer, vérifie d'abord avec `lsof -i :3000` (ou en changeant de port) que l'agent n'a pas laissé tourner sa propre instance : c'est la cause n°1 des faux bugs à cette étape.*

ÉTAPE 14

## Tester en local avant de valider

Une fois que ton agent IA a traité le PRD, il ne pousse jamais directement en production : c'est la règle d'or pour garder une application maintenable. Il te livre une pull request en brouillon sur GitHub, qui reprend ce qui change, ce qu'il a construit et une note explicative. Vérifie d'abord qu'elle n'a aucun conflit avant d'aller plus loin.

Ensuite, passe au test concret sur ta machine :
1. Récupère la dernière version du code avec un pull, pour être sûr de tester exactement ce que l'agent a produit.
2. Lance ton serveur de développement avec `npm run dev` (ou la commande équivalente de ton projet).
3. Ouvre les pages concernées par le PRD et compare le rendu avec la demande d'origine : chaque point du PRD (contexte, correction attendue, fichier ou composant concerné) doit être vérifié un par un.

Piège classique : le port 3000 est déjà occupé, tout simplement parce que l'agent a lancé le serveur lui-même pendant son travail. Tue le process existant ou change de port au lieu de chercher un bug qui n'existe pas.

Autre piège : l'agent peut te dire « c'est fait » sans avoir réellement testé, par exemple s'il n'a pas réussi à ouvrir un navigateur. Ne prends jamais son affirmation pour argent comptant : ouvre toujours la page toi-même. Si le résultat est décevant (modifications cosmétiques ratées, élément non touché), tu as deux options : corriger à la main en local, ou remonter un nouveau ticket via le widget pour alimenter le prochain lot PRD.

Tu sais que c'est bon quand : la PR est sans conflit, l'application tourne en local sans erreur, et chaque demande du PRD est visiblement corrigée à l'écran. À ce moment-là seulement, tu passes le lot en « Review », tu merges la PR et tu clôtures le PRD, avant de promouvoir vers le staging pour la recette client.

*→ Prévois 5 à 10 minutes de test manuel par demande contenue dans le PRD, chronomètre en main. Pour un lot de 2 tickets, bloque 15 minutes : ouvre chaque page modifiée, recharge avec un cache vidé (Ctrl+Shift+R) et vérifie chaque point du PRD un par un. Si le serveur refuse de démarrer parce que le port 3000 est occupé, lance `npx kill-port 3000` au lieu de redémarrer ta machine : c'est presque toujours l'agent qui a laissé son propre serveur tourner.*

ÉTAPE 15

## Merger la PR et promouvoir vers le staging pour la recette client

Une fois que ton lot PRD est marqué « Livré » et que tu as testé la feature en local (avec `npm run dev`), tu passes le PRD en statut « Review » dans ton outil de ticketing. C'est le signal que tu as relu la pull request : vérifie qu'elle reprend bien ce qui change, ce que l'agent a construit, et qu'il n'y a aucun conflit de merge. Chez moi, la PR est créée en brouillon (draft) sur GitHub, donc je la relis calmement avant d'agir. Une fois la review faite, tu merges la PR et tu confirmes : derrière, tu peux clôturer le PRD, et la pull request apparaît bien comme clôturée dans GitHub.

Vient ensuite la promotion vers le staging, l'environnement de recette réservé au client. Tu merges la promotion de dev vers staging, et automatiquement le code est poussé : si tu vas dans les pull requests du dépôt staging, tu la retrouves dans l'onglet « Closed », avec la mention de promotion de dev vers staging. Contrôle aussi l'onglet des commits : tu dois y voir le push effectué sur la branche de staging. Tu sais que c'est bon quand le client peut ouvrir l'URL de staging et tester la feature dans les mêmes conditions que la production, sans toucher à celle-ci.

Point crucial : le client ne valide pas encore la production. S'il dit « non, ça ne va pas en staging », tu cliques sur « Renvoyer une correction » dans l'outil, tu expliques précisément ce qui ne va pas, et vous repartez sur un cycle complet, toujours dans le même PRD, sans en créer un nouveau. Ce n'est qu'après sa validation explicite que tu déploies en production avec un dernier merge. Piège classique : sauter l'étape staging parce que « ça marche en local » ; les différences de code et de données entre environnements font que c'est exactement là que les bugs se cachent.

*→ COMMENT vérifier la promotion : après chaque merge vers staging, ouvre systématiquement deux onglets, l'onglet « Pull requests > Closed » (la PR de promotion dev vers staging doit y figurer) et l'onglet « Commits » (le push sur la branche staging doit être visible avec l'horodatage). COMBIEN de temps laisser au client pour la recette : 48 à 72 heures ouvrées, avec un message type « dis-moi OK ou liste-moi ce qui bloque » ; passé ce délai sans retour, relance-le avant tout déploiement en production.*

ÉTAPE 16

## Pousser en production ou renvoyer une correction dans le même PRD

Dernière étape du cycle : la mise en production, mais avec une porte de sortie. Une fois que le client a testé la feature sur le staging, deux scénarios possibles.

Scénario 1 : le client valide. Tu merges la pull request vers la branche de production, tu confirmes, et la feature part en prod. Dans l'outil, le PRD reste au statut « Livré » et tous les tickets rattachés passent automatiquement en « Livré » aussi : tu n'as rien à déplacer à la main, les colonnes du Kanban ne sont qu'une vue, tout se met à jour tout seul derrière.

Scénario 2 : le client dit « non, ça ne va pas en staging ». Là, tu ne crées surtout pas un nouveau ticket ou un nouveau PRD : tu cliques sur « Renvoyer une correction », tu expliques précisément ce qui ne va pas, et tu repars sur un cycle complet (traitement par l'agent, nouvelle PR, test local, staging) toujours dans le même PRD. C'est ce qui garde l'historique propre : toutes les allers-retours d'une même feature restent regroupés au même endroit, avec les tickets d'origine, les commentaires et les corrections successives.

Une fois la production atteinte, tu génères le changelog : tu crées un nouveau changelog, tu sélectionnes les PRD qui n'en ont pas encore, tu cliques sur « Générer par l'IA », et elle récupère toutes les informations des PRD pour rédiger un résumé destiné au client (par exemple : « Vous pouvez désormais modifier les titres de leçon directement en place ») et utile pour toi aussi, pour savoir où tu en es. Tu publies, et le client est alerté que ses tickets ont été traités.

Tu sais que c'est bon quand : la PR de promotion vers la prod apparaît dans les « Closed » de ton dépôt, le commit de push est visible sur la branche de production, le changelog est publié, et chaque PRD (PRD 6, PRD 7...) affiche bien ses tickets rattachés en statut « Livré ».

Piège courant : pousser directement en prod sans passer par la validation staging du client, ou créer un nouveau PRD à chaque correction au lieu de rester dans le même. Dans les deux cas, tu perds la traçabilité qui rend le projet maintenable dans le temps.

*→ Ne clôture un PRD qu'après validation explicite du client sur le staging, jamais avant. Si le client tarde à répondre, relance-le avec un message ciblé qui liste les 2 ou 3 points précis à tester (par exemple : « vérifie le design du tableau Studio et celui de la page blog ») plutôt qu'un « tu peux tester ? » générique : tu divises le délai de recette par deux et tu évites les validations à moitié faites qui reviennent en correction une fois en prod.*

ÉTAPE 17

## Générer et publier le changelog client

Une fois tes lots PRD livrés et tes tickets clôturés, il reste une dernière étape : informer ton client que ses demandes ont été traitées. C'est le rôle du changelog. Ne le confonds pas avec une note technique interne : le changelog est rédigé pour le client, avec un résumé clair de ce qui a été fait, sans jargon. Voici le process complet :

1. Vérifie que le statut de tes lots est bien « Livré ». Un changelog ne se crée qu'après la mise en production effective, jamais avant : sinon tu annonces des features que le client ne peut pas encore voir, et tu crées de la confusion.

2. Crée un nouveau changelog dans ton outil de ticketing. Sélectionne les PRD qui n'ont pas encore de changelog rattaché. Dans l'exemple du process, on sélectionne le PRD 6 (« modifier les titres de leçon directement en place ») et le PRD 7 (« amélioration du design du tableau ») en une seule fois.

3. Lance la génération par l'IA. L'assistant récupère automatiquement toutes les informations des PRD sélectionnés (contexte, tickets associés, corrections appliquées) et propose une description destinée au client, mais aussi à toi, pour que tu saches où tu en es. Le résultat ressemble à : « Vous pouvez désormais modifier les titres de leçon directement en place, sans ouvrir... ».

4. Relis et ajuste la formulation. L'IA rédige le brouillon, mais tu restes responsable du ton : phrases courtes, bénéfices avant fonctionnalités, zéro terme technique (jamais « merge de la PR » ou « refactor du composant »). Le client doit comprendre en 30 secondes ce qui a changé pour lui.

5. Publie. Chaque changelog conserve la trace des PRD et des tickets qu'il contient : tu peux cliquer sur chacun pour voir à quoi il correspond. Côté tickets, tout est passé en « Livré » automatiquement : ce n'est plus à toi de les déplacer, la vue se met à jour toute seule.

Tu sais que c'est bien fait quand le client peut lire ton changelog sans te poser de question, et que toi, en interne, tu retrouves en un clic quels tickets ont été traités dans quelle version. Piège classique : laisser des PRD sans changelog pendant des semaines. Au bout de trois lots en attente, tu ne sais plus ce qui a été communiqué au client, et lui ne sait plus ce qui a été corrigé.

*→ Génère ton changelog immédiatement après chaque déploiement en production, le jour même, jamais à la fin du mois. Le réflexe concret : dès que le merge prod est confirmé, tu ouvres ton outil, tu sélectionnes les PRD « Livré » sans changelog, tu cliques sur « Générer par l'IA », tu relis 2 minutes et tu publies. Compte 5 à 10 minutes maximum par changelog. Si tu factures un client à la recette, ce délai de 24 h entre livraison et annonce est aussi un signal de professionnalisme qui justifie tes tarifs.*

ÉTAPE 18

## Boucler la boucle : tickets livrés et statuts synchronisés

Dernière étape, et pas la moindre : faire remonter automatiquement l'état d'avancement dans ton outil de tickets, pour que rien ne dépende de ta mémoire. Le principe : c'est l'agent IA qui met à jour les statuts, pas toi. Tu ne déplaces plus jamais un ticket à la main dans le kanban. Voici le déroulé complet.

1. Passage en « En cours ». Quand tu lances ton plugin de traitement (chez moi, « ticket dev ») en local, il lit le PRD récupéré via un pull, puis modifie le fichier `index.json` de l'outil de ticketing pour indiquer qu'il démarre. Si tu rafraîchis l'interface, le lot PRD est passé de « PRD » à « En cours ». Signal de réussite : le statut change sans que tu aies touché quoi que ce soit dans l'interface.

2. Passage en « Livré » et création de la pull request. Une fois les modifications terminées, l'agent teste (il a besoin d'un navigateur pilotable, sinon il te dira qu'il n'a pas pu tester), met à jour le statut dans l'outil et ouvre automatiquement une pull request en brouillon (draft) sur GitHub. La PR reprend ce qui change, ce qui a été construit et la note de synthèse. Tu la retrouves aussi dans l'onglet « PR » de ton outil, rattachée au bon PRD. Vérifie qu'il n'y a aucun conflit avant d'aller plus loin.

3. Review et clôture. Tu testes en local avec `npm run dev` (piège classique : le port 3000 est parfois déjà occupé parce que l'agent a lancé le serveur lui-même, tue le process avant de relancer). Si le résultat te convient, tu passes le lot en « Review », tu merges la PR, puis tu clôtures le PRD. Derrière, tu promeus vers le staging : le merge crée une seconde PR de promotion dev → staging, visible dans les PR « Closed » et dans les commits.

4. Boucle de correction ou production. Si le client refuse en recette, tu cliques sur « Renvoyer une correction », tu expliques ce qui ne va pas, et le cycle repart dans le même PRD, sans en créer un nouveau. Si le client valide, tu merges vers la production. À ce stade, tous les tickets du lot passent automatiquement en « Livré » : le kanban devient une simple vue, la synchronisation se fait toute seule en arrière-plan.

5. Génération du changelog. Une fois les lots livrés, tu crées un changelog, tu sélectionnes les PRD qui n'en ont pas encore, et tu cliques sur « Générer par l'IA ». Elle récupère les informations de chaque PRD et rédige un résumé destiné au client (par exemple : « Vous pouvez désormais modifier les titres de leçon directement en place, sans ouvrir l'éditeur »), puis tu publies. Le changelog sert aussi d'historique interne : tu sais exactement quel PRD contenait quels tickets.

*→ Automatise les changements de statut via un simple fichier JSON (`index.json`) que ton agent met à jour à chaque phase : démarrage, livraison, clôture. Ton interface lit ce fichier et affiche le kanban en conséquence. Règle d'or : interdis-toi de déplacer un ticket à la main. Si un statut est faux, c'est que le process a un trou, corrige le process, pas le ticket. Et côté navigateur de test, configure ton agent pour utiliser un outil pilotable fiable (type Dev Browser) plutôt que l'extension Chrome par défaut, sinon il échouera silencieusement à tester avant de livrer.*

---

### Récapitulatif

| Étape | Action clé | Résultat |
|---|---|---|
| 1 | Structurer projets et environnements | Base propre et maintenable |
| 2 | Collecter les tickets via le widget | Demandes capturées sans effort |

| | | |
|---|---|---|
| 3 | Regrouper en lot PRD | Cahier des charges prêt pour l'IA |
| 4 | Livrer le PRD à l'agent IA | Code modifié, PR créée automatiquement |
| 5 | Tester, merger, promouvoir | Feature validée en recette puis en prod |
| 6 | Générer le changelog | Client informé, boucle bouclée |

Ce process, c'est exactement le genre de système qu'on construit ensemble quand tu veux intégrer l'IA dans ton workflow sans perdre le contrôle. Si tu veux structurer ta façon de travailler avec l'IA, parlons-en.