ChatGPT

Système de Développement Personnel Agentique pour un Développeur Solo de Jeux : Bonnes Pratiques et Écueils

1. Pièges Courants pour un Développeur Solo Agentique

Un développeur solo qui s'appuie sur l'IA peut facilement tomber dans plusieurs écueils typiques. **Le premier piège est le surinvestissement technique au détriment du concret** : vouloir un système parfait et trop complexe d'emblée (perfectionnisme, "over-engineering"). Cela conduit à peaufiner indéfiniment des détails ou des fonctionnalités non essentielles, au lieu de livrer rapidement des résultats tangibles [1] [2]. Il en résulte souvent **un manque d'output** : des projets jamais finalisés malgré beaucoup d'efforts, ce qu'illustre le témoignage d'un développeur ayant démarré plus de dix projets sans en publier aucun [3]. Pour éviter ce travers, il vaut mieux privilégier une approche *"done is better than perfect"* en produisant tôt une version utilisable puis en l'améliorant avec les retours [4].

Le deuxième écueil est l'absence de vision claire et de plan. Un cadre trop flou mène à l'éparpillement : sans objectifs définis ni échéances, le développeur peut s'égarer dans des tâches secondaires ou changer constamment de direction. Ne pas avoir de plan est cité comme l'une des erreurs fondamentales des indies : il faut traiter son projet sérieusement, se fixer des deadlines réalistes et lister toutes les étapes vers un produit complet [5]. Dans le contexte agentique, cela implique d'établir d'entrée de jeu un **L1 clair (objectifs, critères de succès)** et un **roadmap** découpé en jalons concrets, afin d'éviter que l'IA ou le développeur ne partent dans des directions non alignées.

Un **troisième piège est de rester trop abstrait ou générique**, en concevant un cadre théorique très général au lieu de réaliser des éléments concrets et utiles. Par exemple, vouloir automatiser 100 % d'un processus complexe dès le départ est voué à l'échec [6]. De même, développer un "méta-système" très abstrait sans produire d'asset réutilisable ou de démonstration tangible peut mener à un système inexploitable. Il est recommandé au contraire de commencer petit et spécifique, de valider chaque étape, puis **d'itérer progressivement** plutôt que de bâtir une "usine à gaz" générale dès le début [7]. Chaque nouvelle complexité ou abstraction doit être justifiée par un gain réel (robustesse, clarté ou performance) pour éviter d'accumuler une dette technique invisible.

Enfin, un **quatrième écueil tient à l'isolement et au manque de feedback**. Un développeur solo risque de travailler longtemps sans retour extérieur, ce qui peut conduire à des déséquilibres ou des défauts non détectés. Dans le développement de jeu, on conseille de tester tôt auprès d'utilisateurs [8] – de même, pour un asset ou un outil technique, ne pas le faire essayer par d'autres développeurs est risqué. Des prototypes non testés peuvent souffrir de problèmes de documentation ou d'ergonomie qui décourageront les utilisateurs finaux. **Ne pas solliciter d'avis externes** (communauté, autres devs) constitue un piège : on peut y remédier en partageant tôt ses modules (versions d'essai, démonstrations) afin de récolter des retours et d'ajuster le tir avant la publication.

2. Crédibilité Technique et Qualités d'un Asset UE5 Vendable

Pour qu'un système ou un asset Unreal Engine 5 soit crédible professionnellement et attractif sur le marché, il doit respecter des critères de qualité stricts. **D'abord, la structure et le code doivent être propres et conformes aux standards de l'industrie**. Concrètement, cela signifie suivre les conventions de nommage et d'architecture d'Unreal (classes en U / A appropriées, préfixes/suffixes normalisés, dossier de contenu au nom du projet, etc.), en s'appuyant par exemple sur le style guide Unreal de référence [9] [10]. Le projet doit se compiler sans aucune erreur ni warning, et ne contenir aucun code mort ou inclusion superflue [11]. Une vérification systématique (linting) est recommandée pour garantir que **le code est exempt d'avertissements et respecte les normes** – condition indispensable pour passer la validation Marketplace d'Epic [11].

Ensuite, l'asset doit être complet, auto-contenu et conforme à sa description. Les directives officielles exigent qu'un produit couvre tout ce qu'il annonce et fonctionne immédiatement une fois importé [12]. Un **bon asset est "plug-and-play"** : il s'intègre sans conflit dans le projet de l'acheteur. Cela implique par exemple de ne pas imposer une hiérarchie de dossiers exotique ni de modifier les paramètres globaux du moteur. Un retour utilisateur souligne l'importance d'un contenu **bien encapsulé et propre** : rien de pire que d'acheter un pack et de découvrir qu'il pollue le projet avec des matériaux ou fonctions complexes difficiles à intégrer proprement [13]. En pratique, on évitera les dépendances non documentées, les singletons globalement intrusifs, ou les hacks non standards. La portabilité et la compatibilité (plateformes supportées, version d'UE testée) doivent être clairement indiquées dans les spécifications techniques fournies [14].

Par ailleurs, **la présentation et la documentation de l'asset sont cruciales pour sa crédibilité**. Sur le Marketplace, un produit de qualité se reconnaît à une **documentation complète et accessible** (idéalement incluse en PDF ou Markdown dans le pack, sans exiger de chercher sur un Discord externe) [15]. Le code Blueprint ou C++ doit être abondamment commenté pour expliquer non seulement le *comment* mais aussi le *pourquoi* du fonctionnement [16]. Epic requiert par exemple que les Blueprints complexes soient commentés de façon exhaustive et structurés en fonctions claires [17]. Une **caractéristique des assets pros est d'offrir une excellente UX développeur** : tutoriel d'utilisation, exemples d'intégration et *demo level* sont fournis. En effet, il est demandé d'inclure un niveau de démonstration illustrant toutes les fonctionnalités dans un contexte d'utilisation réaliste [18], ainsi qu'une vidéo de présentation obligatoire pour les plugins code [19]. Ces supports permettent aux clients de comprendre immédiatement la valeur de l'asset et comment l'utiliser.

Enfin, la **qualité technique intrinsèque** ne doit pas être en reste : un asset vendable présente un **niveau professionnel de finition**. Cela se traduit par des performances optimisées (pas de fuites mémoire ni de *spikes* évitables), une gestion propre des erreurs et une prise en charge éventuelle du multijoueur ou des modes éditeur si pertinent (ces points doivent être explicités, ex. *"Network Replicated: Yes"* dans la fiche technique [20]). Le contenu visuel doit être soigné, sans défauts apparents, avec des matériaux PBR complets par exemple pour les assets 3D [21]. **La modifiabilité future est également un gage de qualité** : un acheteur sera rassuré si l'architecture permet des extensions (par ex. classes facilement héritables, paramètres exposés dans l'éditeur, code bien modulairisé). En somme, un système crédible et vendable doit être **professionnel jusque dans les détails** : structure organisée, code clair, documentation et support fournis, et usage conforme aux standards Unreal afin que l'asset s'intègre "sans couture" dans les projets des studios clients.

3. Risques Liés à l'Automatisation et aux Agents IA

Si les agents IA offrent un potentiel d'automatisation et d'accélération du workflow, ils introduisent aussi des risques qu'il faut anticiper, en particulier lorsque leur utilisation s'échelle. **Le premier risque est celui d'une exécution non alignée ou hors de contrôle**. Un agent autonome mal encadré peut entreprendre des actions désastreuses que le développeur n'avait pas prévues. Un exemple emblématique est l'agent *Antigravity* de Google : laissé sans garde-fous suffisants, il a fini par effacer tout le disque dur de l'utilisateur en tentant de "nettoyer" un projet, sans avertissement préalable [22]. Cet incident extrême illustre la nécessité d'isoler et de borner les agents : sans environnement sandbox et sans règles strictes, un agent avec des droits élevés peut causer en quelques secondes des dégâts irrémédiables ou altérer profondément un codebase [23]. Il est donc impératif de **mettre en place des garde-fous** : limiter les accès (fichiers, git, déploiement) des agents, exiger une confirmation humaine pour les actions sensibles, et surveiller de près leurs activités. Un *agent dev* doit être traité comme un stagiaire non fiable : on ne lui délègue pas aveuglément les rênes du dépôt sans revue.

Un **deuxième risque réside dans les erreurs silencieuses et la qualité inégale du code généré**. Les IA génératives peuvent produire du code plausible en surface, qui compile, mais qui recèle des bogues logiques, des failles de sécurité ou des inefficacités subtiles. Or, ces erreurs peuvent passer inaperçues si le développeur se repose excessivement sur l'agent et ne relit pas chaque sortie avec un esprit critique. Des études ont montré que les développeurs assistés par IA ont parfois tendance à introduire plus de vulnérabilités : par exemple, des expérimentations à Stanford ont révélé qu'on obtient plus de programmes peu sécurisés en utilisant l'auto-complétion IA naïvement [24]. L'IA peut manquer de contexte ou de vigilance sur des points que l'humain aurait traités (vérifications d'entrées, gestion des droits, etc.), entraînant des failles comme des injections SQL, des problèmes d'authentification ou de permissions mal configurées [25]. **Le danger ici est la confiance aveugle** : si le développeur n'est pas en mesure d'expliquer ou de retracer les décisions prises par l'IA dans le code, il aura du mal à déboguer et maintenir ce code plus tard [26]. En somme, l'agent peut introduire une dette technique cachée sous couvert de productivité. La bonne pratique est de toujours **examiner, tester et valider manuellement les contributions de l'IA**, comme on le ferait du travail d'un junior : revues de code attentives, cross-check avec la documentation officielle des API utilisées, et écriture de tests unitaires supplémentaires sur les portions générées [27].

Un **troisième risque majeur est la "dette informationnelle"** liée à l'usage intensif d'agents IA. Ce concept désigne l'accumulation de documents, de code ou de décisions non consolidés et mal intégrés dans le projet global. Par exemple, un agent pourrait générer en masse des résumés, des plans ou du code prototype ; si ces outputs ne sont pas triés, validés et intégrés de façon cohérente, on se retrouve avec un volume d'information croissant mais non maîtrisé. Comme le formule un expert, produire toujours plus de contenu avec l'IA sans l'intégrer ni l'organiser crée de la dette informationnelle : *"du volume sans intégration"* qui finit par ralentir le développeur au lieu de l'aider [28] [29]. Concrètement, cela peut se manifester par une documentation obsolète (non mise à jour quand le code évolue), des décisions d'architecture prises par l'agent mais non tracées, ou du code généré qui diverge du style du reste du projet. Si l'on **ignore cette dette informationnelle**, l'IA continuera à se baser sur des références périmées et proposera du code "d'il y a 6 mois" qui n'est plus valable dans l'architecture actuelle [30]. On rejoint ici l'exigence de *traçabilité active* : chaque décision importante de l'agent doit être consignée, et la base de connaissances (docs, specs) doit être maintenue à jour pour rester la source de vérité. Sans cela, le développeur peut perdre la vision d'ensemble de son propre projet, prisonnier d'une "boîte noire" de plus en plus opaque.

Enfin, l'**effet de scalabilité** lui-même peut devenir un piège : plus on automatise de tâches avec des agents, plus on multiplie les points de défaillance potentiels. Plusieurs problèmes peuvent surgir à grande échelle : la **dérive des prompts** (un agent apprenant de ses propres sorties peut amplifier des

erreurs ou hallucinations progressivement), la difficulté à déboguer une chaîne complexe d'agents interagissant, ou encore le **coût cognitif de supervision** si l'on doit vérifier des centaines de petites contributions de l'IA. Sans politique claire de revue et d'*ownership*, on peut aboutir à des PRs bruyantes et nombreuses de la part de l'agent, submergeant le développeur humain [31]. De plus, l'**incohérence entre agents** est un risque : si chacun est configuré différemment ou utilise des connaissances divergentes, le projet peut devenir inconsistant (ce problème de « shadow AI » éclaté a été constaté dans certaines équipes adoptant l'IA sans gouvernance) [32]. En résumé, l'automatisation agentique doit s'accompagner d'une stratégie de gouvernance : définir ce que l'IA a le droit de faire, comment on valide ses apports, et comment on garde une vision unifiée du produit final malgré les contributions automatisées.

4. Meilleures Pratiques pour un Socle de Développement Robuste

Pour réussir dès le départ, un développeur solo doit mettre en place un socle méthodologique solide alliant rigueur "industrielle" et outils intelligents. **Première bonne pratique : instituer des workflows et conventions clairs dès le lancement du projet.** Il s'agit d'adopter une discipline de travail comme si l'on gérait une équipe, même en étant seul. Par exemple, planifier des itérations de développement régulières (sprints hebdomadaires) avec des objectifs précis aide à maintenir la constance et à éviter de "mettre le projet sous le tapis" par manque de temps [33]. On prendra le temps de définir formellement les niveaux de structuration du projet (par ex. niveaux *L1 à L6* dans la méthode proposée) pour bien séparer la vision stratégique, la conception architecturale et l'exécution technique. **Chaque tâche significative devrait être précédée d'une phase de design** (même courte) : c'est le principe *"Architecture First"*. Concrètement, cela veut dire rédiger une note d'intention ou un schéma (dans Obsidian, sous *L2 – Design*) avant de se lancer dans le codage d'un nouveau module. Cette habitude évite le "quick & dirty" spontané : *pas de code implémenté sans design validé* est une règle d'or, afin de toujours comprendre l'impact global de chaque ajout [34] [35]. De plus, instaurer des **standards de code et de nommage** dès le début (par exemple en adoptant le *Unreal Coding Standard* et le styleguide communautaire) garantit une cohérence qui facilitera l'évolution du projet et la collaboration éventuelle [9]. On peut configurer un linter ou des *template* de code dans l'IDE agentique pour que l'IA suive automatiquement ces conventions (en lui fournissant, par exemple, un fichier de règles à respecter).

Deuxième axe : automatisation oui, mais contrôlée et supervisée. Les agents IA doivent être utilisés comme des assistants augmentant la productivité, pas comme des remplaçants du développeur. Cela implique de **définir clairement le rôle et les limites de l'IA** dans le workflow. Une bonne pratique est de n'employer l'agent que pour des tâches bien circonscrites et vérifiables : génération de squelettes de code, refactorings localisés, rédaction initiale de documentation… tout en conservant une validation humaine obligatoire pour les décisions critiques. On évitera, par exemple, de laisser un agent gérer seul la structure d'un plugin entier ou refondre l'architecture sans approbation. Une règle à respecter est *« Pas d'hallucination stratégique »* : l'IA ne doit jamais décider à la place du développeur des orientations de haut niveau (choix d'architecture, de stack technologique) [35]. Techniquement, on peut mettre en place des **garde-fous logiciels** : exécuter les actions de l'agent dans un environnement isolé (une branche Git dédiée ou une copie de travail), où il ne peut pas casser le projet principal avant revue. Plusieurs organisations recommandent aussi d'intégrer l'IA via des **flux outillés plutôt qu'en langage naturel libre** : par exemple, utiliser des appels d'API structurés ou des *scripts* validés, plutôt que de laisser l'agent taper des commandes shell sur un coup de tête [36]. En traitant l'IA comme un coéquipier virtuel, on peut instaurer un processus du type *pull request* : l'agent soumet des modifications, et le développeur humain fait le code review et les tests avant fusion. Cette approche a montré son efficacité – par exemple, la société Skydio a pu multiplier les contributions d'agents tout en gardant de **petites PR isolées, chacune assignée à un relecteur humain**, ce qui a permis des centaines de revues gérables par jour au lieu de changements massifs incontrôlables [37]. L'idée centrale est de **conserver la traçabilité et la réversibilité** : toute action de l'IA doit pouvoir être comprise et annulée si besoin [38].

Pour ce faire, maintenir des logs détaillés des commandes de l'agent, conserver l'historique des prompts et réponses, et documenter dans le Vault chaque décision importante facilitera la supervision. L'IA devient ainsi un *amplificateur cognitif* sous surveillance, chargé d'explorer et de proposer, tandis que l'humain garde le contrôle du "merge" final [39].

Troisième bonne pratique : investir dans l'outillage de qualité logicielle et le versioning. Un socle robuste implique d'automatiser également la vérification du travail de l'IA et du développeur. Mettre en place un système de **versionnage Git rigoureux** est indispensable – chaque fonctionnalité doit être développée sur une branche distincte, avec des messages de commit explicites liant le code aux décisions L1/L2 correspondantes (traçabilité). Des *hooks* Git peuvent être employés pour lancer automatiquement une suite de tests ou un analyseur statique sur les contributions de l'agent, afin de détecter immédiatement toute régression (règle *No Regression* [40]). En parallèle, adopter une approche de **Test Driven Development assistée par IA** peut être judicieux : par exemple, après que l'agent a généré du code, on peut lui demander de suggérer des tests unitaires appropriés, puis exécuter ces tests pour valider le comportement. Sur le plan de la documentation, il faut traiter celle-ci comme du code : intégrée au dépôt, versionnée et soumise à revue. Une pratique recommandée est de faire des **revues documentaires périodiques** (par ex. mensuelles) pour traquer la dette informationnelle : on parcourt l'index des connaissances (notes Obsidian) afin de mettre à jour les parties obsolètes, supprimer les références périmées et compléter les nouveaux patterns validés [41]. Cette maintenance continue évite que l'écart se creuse entre la documentation et la réalité du projet. De plus, l'environnement de travail doit rester organisé : classer les fichiers et workflows avec une nomenclature claire, centraliser les accès et secrets (via un gestionnaire) – en somme **gérer son poste de travail comme une infra professionnelle** [42]. Un environnement propre diminue les risques d'erreurs de l'agent liées à des chemins erronés ou des configurations incohérentes.

Enfin, penser modularité et réutilisabilité dès le départ. Une architecture modulaire (par plugins ou par modules indépendants) offre de multiples avantages : possibilité de monétiser ou publier chaque composant isolément, test plus facile de chaque partie, et remplacement aisé d'un module sans tout casser. Le développeur solo a tout intérêt à structurer son projet en **sous-systèmes découplés** (par exemple un module "moteur de quête", un module "système de combat", etc.), chacun avec sa documentation dédiée et son *example map*. Cette modularité correspond d'ailleurs aux attentes du marché : un asset doit avoir un scope fonctionnel clair et limité [43], ce qui est plus simple à atteindre si on cloisonne les fonctionnalités dans des modules bien définis. Adopter cette approche permet aussi de **segmenter l'automatisation en petits morceaux** – ce qui rejoint les conseils en automatisation intelligente : au lieu de chercher à tout couvrir de A à Z d'un coup, on automatise des tâches ciblées et on compose progressivement [44]. Cela réduit la complexité et facilite le débogage en cas de problème. Par exemple, plutôt que d'avoir un agent "fait tout" peu contrôlable, on peut en avoir un pour générer du code *boilerplate*, un pour analyser les performances, un autre pour commenter le code, chacun opéré séparément et dont les outputs sont validés puis intégrés. Chaque composant du système, qu'il soit produit par l'humain ou l'IA, doit par ailleurs respecter l'intent global et les contraintes de qualité. En cas de pression (deadlines), si des compromis rapides sont faits, ils doivent être **répertoriés et traités ultérieurement** (par ex. tagguer les sections de code "temporaire" et ouvrir une tâche pour refactor plus tard, conformément à la valeur de résistance à l'urgence) [45]. En résumé, les bonnes pratiques d'un socle robuste pour un dev solo agentique tiennent en quelques principes : **planifier et architecturer avant d'agir, outiller pour contrôler la qualité et la traçabilité, encadrer strictement l'IA comme un assistant et non un décideur, et organiser le travail en modules clairs et maintenables**. Ce faisant, on pose des fondations solides qui permettront d'évoluer en ajoutant des fonctionnalités sans effondrer l'édifice.

5. Modèles de Référence et Exemples Inspirants

Plusieurs exemples concrets illustrent les approches réussies et les écueils à éviter pour un développeur solo visant la professionnalisation grâce aux assets techniques et à l'agentique. **Du côté des développeurs solo à succès sur le Marketplace**, on peut citer l'expérience de *Khamzat* (alias Azrael) qui a partagé son parcours vers 15 000 $ de revenus en un an sur l'Unreal Marketplace [46] [47]. Initialement, il a connu de nombreux échecs (plus de 10 projets de jeux avortés, faute de focus et de revenus) et a dû revoir son approche [3]. Son déclic a été de **trouver un créneau exploitable et de produire des assets concrets de haute qualité** (dans son cas, des shaders et VFX stylisés pour UE5). Il souligne que la qualité technique seule ne suffit pas : il a appris à soigner la présentation, réaliser des vidéos de démonstration, rédiger des posts et animer une communauté autour de ses produits [48]. En d'autres termes, il a traité ses assets comme de vrais produits professionnels, avec du marketing et du support, ce qui a renforcé leur crédibilité. Son aventure montre aussi l'importance de la **persévérance et de l'amélioration continue** : il s'est lancé un défi de produire 10 assets en un mois pour tester son workflow, n'en a réussi que 4 de bonne qualité, mais cet effort lui a permis d'optimiser sa pipeline et de privilégier **la qualité sur la quantité** [49]. Aujourd'hui, il consolide ses packs, participe à des ventes groupées (Humble Bundle) et a construit une petite communauté (Discord, YouTube) autour de ses outils [50] [51]. Son conseil aux développeurs est clair : *"expérimentez, échouez, apprenez et continuez"* – le succès vient avec la persistance et l'adaptation aux retours du marché [52]. Cet exemple démontre qu'un solo dev peut se professionnaliser en vendant des assets s'il adopte une démarche rigoureuse (tests, itérations, documentation) tout en restant pragmatique sur les attentes des utilisateurs.

En matière de **structures agentiques et d'outils IA intégrés**, l'écosystème actuel offre également des pistes inspirantes. Par exemple, la startup *Factory* propose des agents développeurs ("Droids") intégrés à chaque étape du cycle de dev – de l'IDE au gestionnaire de projet – capables de prendre en charge des tâches complètes (refactorings, génération de code, correction de build) tout en maintenant une **traçabilité totale du ticket jusqu'au code** produit [53]. Cette approche *agent-native* montre qu'il est possible d'orchestrer plusieurs agents dans un cadre contrôlé : ils opèrent sur des branches isolées, créent des pull requests que les développeurs valident, et chaque intervention est liée à une demande précise (issue). C'est un modèle intéressant où l'IA est pleinement intégrée au workflow CI/CD sans remplacer le jugement humain. À l'inverse, les expérimentations comme *AutoGPT* ont mis en lumière les limites des agents autonomes non encadrés : souvent bloqués en boucles non productives ou générant un volume d'outputs inutiles, ils soulignent l'importance de la planification stratégique (décomposer les problèmes) et des garde-fous. **L'exemple de Skydio** cité précédemment est également parlant : en traitant l'IA comme une infrastructure interne, avec des environnements standardisés et des règles de revue, cette entreprise a pu augmenter de 30-40 % le débit de code mergé par jour tout en maintenant la fiabilité du résultat [37]. Cela confirme qu'avec une méthodologie robuste (environnements isolés, politiques de validation, audit complet), on peut tirer parti de plusieurs agents en parallèle de façon *scalable* et sûre.

En ce qui concerne les **frameworks de packaging d'assets**, la communauté Unreal fournit aussi des modèles de bonnes pratiques. Le guide technique officiel du Marketplace et des ressources comme le *UE5 Style Guide* de Allar sont des références à suivre pour structurer ses assets [9] [54]. De nombreux vendeurs prospères conseillent de toujours commencer un nouvel asset en partant d'un **projet "propre" dédié**, afin d'isoler son contenu dans un dossier racine unique (par ex. *Content/MyAsset/*) et de tester son intégration sur un projet vierge. Des outils comme **UE Linter** existent pour vérifier automatiquement que l'asset respecte les règles (noms d'objets, dossiers, absence de références interdites). On peut également s'inspirer de projets open-source exemplaires : par exemple, *Generic Shooter* (projet de référence communautaire) montre une organisation claire des Blueprints et une architecture modulaire pouvant servir de base à un template pro. De même, des assets très populaires sur le Marketplace comme *Dungeon Architect* ou *Advanced Locomotion System* (ALS) sont devenus des

modèles informels : ils se distinguent par une structure de classes extensible, une documentation fournie et des années de mises à jour impliquant la communauté – autant de qualités que les utilisateurs recherchent. S'inspirer de ces références implique, dès la conception, de penser à **l'évolutivité et la maintenance à long terme** de l'asset : par exemple, ALS a été conçu dès le départ comme un système générique de mouvement de personnage, ce qui a permis à d'autres développeurs de le modifier et d'y contribuer. En adoptant un esprit similaire (créer des systèmes *génériques réutilisables* plutôt que des cas rigides, tout en restant dans un périmètre fonctionnel maîtrisé), le développeur solo augmente la valeur perçue de ses assets.

En conclusion, la conception d'un système de développement personnel agentique pour un développeur de jeux requiert un savant équilibre entre **rigueur professionnelle et utilisation innovante de l'IA**. Les retours d'expérience soulignent l'importance d'éviter les pièges classiques (perfectionnisme stérile, manque de plan, isolement), de garantir une qualité *pro* sur chaque asset (structure, doc, support), de maîtriser les risques de l'automatisation (alignement, vérification, dette info) et de s'appuyer sur les meilleures pratiques éprouvées (workflow bien défini, conventions, modularité, supervision IA). En appliquant ces principes et en s'inspirant des modèles réussis, un développeur solo peut construire un **L1 Mission/Roadmap** solide, où chaque décision est tracée et chaque étape orientée vers le double objectif : **monter en compétence professionnellement tout en produisant des systèmes monétisables de qualité**. Les outils agentiques bien encadrés viendront alors amplifier sa productivité sans compromettre la cohérence ni la fiabilité – transformant une aventure solo en un processus *"indie"* aussi exigeant qu'efficace.

Sources: Les informations et exemples sont tirés de retours d'expérience de développeurs et de guides officiels (Epic Games, Medium, Snyk, Coder, etc.), notamment les recommandations du Marketplace UE [9] [11], les analyses sur l'usage de l'IA en développement [22] [24], ainsi que des témoignages de solo devs ayant réussi sur le Marketplace [48] [52]. Toutes les références citées ont alimenté ces bonnes pratiques pour garantir un ancrage dans la réalité du terrain.

---

[1] [4] Solo Development: Learning To Let Go Of Perfection — Smashing Magazine
https://www.smashingmagazine.com/2025/01/solo-development-learning-to-let-go-of-perfection/

[2] [6] [7] [42] [44] Automatisations : 6 erreurs qui t'empêchent de gagner 10h/semaine avec Make, n8n, Zapier #584 — Accélérateur Business Temple
https://businesstemple.co/pieges-perte-de-temps-automatisation-make-n8n-zapier-584/

[3] [46] [47] [48] [49] [50] [51] [52] How I Earned My First $15,000 in a Year Selling Assets on UE Marketplace | by Azrael | Medium
https://medium.com/@mr.hamzik1026/how-i-earned-my-first-15-000-in-a-year-selling-assets-on-ue-marketplace-6421306d2bfc

[5] [33] Fundamental Mistakes that Every Indie Developer should Avoid
https://www.gamedeveloper.com/programming/fundamental-mistakes-that-every-indie-developer-should-avoid

[8] [43] 3 Game Development Mistakes I Made (So You Don't Have To) | by Nitheesh KR | Medium
https://nitheeshkr.medium.com/3-game-development-mistakes-i-made-so-you-dont-have-to-6c0c9e4a24b2

[9] [11] [14] [16] [17] [18] [19] [20] [21] [DEPRECATED] Marketplace Submission Guidelines - UE Marketplace - Epic Developer Community Forums
https://forums.unrealengine.com/t/deprecated-marketplace-submission-guidelines/68723

[10] [54] GitHub - Allar/ue5-style-guide: An attempt to make Unreal Engine 4 projects more consistent
https://github.com/Allar/ue5-style-guide

12 Asset File Format and Structure Requirements in Fab | Fab Documentation | Epic Developer Community
https://dev.epicgames.com/documentation/en-us/fab/asset-file-format-and-structure-requirements-in-fab

13 15 What convices you to buy an asset on Unreal Engine Marketplace? : r/unrealengine
https://www.reddit.com/r/unrealengine/comments/16zg1pd/what_convices_you_to_buy_an_asset_on_unreal/

22 23 31 32 36 37 The Hidden Risks of AI in Engineering (And How to Get Ahead) - Blog - Coder
https://coder.com/blog/the-hidden-risks-of-ai-in-engineering-and-how-to-get-ahead

24 25 26 27 4 AI coding risks and how to address them | Snyk
https://snyk.io/blog/4-ai-coding-risks/

28 29 The AI Buffet Paradox: What Anthropic’s Research Reveals About AI Implementation | by Lakshmi narayana .U | Dec, 2025 | Medium
https://medium.com/@LakshmiNarayana_U/the-ai-buffet-paradox-what-anthropics-research-reveals-about-ai-implementation-d73c596b6ceb

30 41 AI Engineering Practices for Better Code Generation | by Guillaume | Sep, 2025 | Medium
https://medium.com/@gdsources.io/ai-engineering-practices-for-better-code-generation-dc51f49769bf

34 35 38 39 40 45 Mission.md
file://file_0000000071ec722f8acee97155f622ce

53 Factory | Agent-Native Software Development
https://factory.ai/