ChatGPT

Conception d'un Agentic Command Center pour un développeur solo Unreal Engine

Ce document propose une analyse stratégique de la conception d'un **Agentic Command Center** : un environnement orchestrant des agents d'IA pour assister un développeur indépendant dans un projet Unreal Engine (via l'IDE agentique *Antigravity* de Google, avec Obsidian comme base documentaire). Nous détaillons les fonctions d'agent essentielles, les règles de gouvernance minimales à chaque couche du système (L0–L6, Lω), la structuration de workflows en phases distinctes, un modèle de documentation distribuée, les anti-patterns critiques à éviter et les bonnes pratiques émergentes pour la supervision de ces agents. L'objectif est d'établir un cadre **actionnable** alliant efficacité et sûreté, en s'appuyant sur des retours d'expérience concrets de la communauté.

1. Fonctions d'agent essentielles pour un développeur solo

Un développeur solo peut déléguer aux agents un éventail de tâches couvrant tout le cycle de développement logiciel. Les fonctions d'agent jugées réellement utiles incluent :

- **Brainstorming & Conception initiale :** un agent peut aider à *brainstormer* des idées, détailler les exigences et explorer différentes approches d'implémentation. Par exemple, de nombreux développeurs commencent désormais par discuter du problème avec l'IA pour clarifier le cahier des charges et les cas limites avant de coder [1]. Cette étape aboutit souvent à un document de spécifications complet (*spec.md*) listant les besoins, le design architectural, les modèles de données et même une stratégie de tests [2].

- **Planification de projet :** un agent peut ensuite élaborer un plan d'action structuré à partir des spécifications. Il s'agit de découper le développement en tâches ou étapes logiques, un peu comme un plan de projet agile. L'agent sert de *planificateur*, suggérant un ordre d'implémentation par petites itérations [3]. Le développeur peut itérer avec l'agent sur ce plan (ajouts, critiques, refinements) jusqu'à obtenir une feuille de route cohérente et exhaustive avant de passer au codage effectif [4].

- **Implémentation du code (agent *coder*) :** c'est le rôle classique de l'AI pair programmer. L'agent prend en charge la rédaction du code pour une tâche donnée du plan, en respectant le style du projet et les contraintes techniques. Il peut créer de nouveaux modules, modifier du code existant, générer des assets Unreal Engine, etc. Dans *Antigravity*, un agent peut opérer de façon autonome sur plusieurs outils (éditeur, terminal, navigateur) pour réaliser une feature de bout en bout [5]. Par exemple, l'agent code la fonctionnalité, lance le jeu pour tester, puis ouvre le navigateur intégré pour vérifier que le composant fonctionne, le tout sans intervention humaine directe [5].

- **Revue de code & Debug :** un agent peut agir en tant que relecteur de code (code reviewer) ou *debugger*. Il analyse les *diffs*, détecte des erreurs ou des mauvaises pratiques, et propose des corrections. Mieux, il peut exécuter la suite de tests automatiquement après chaque implémentation et corriger le code si des tests échouent [6] [7]. Ce **feedback loop** (coder → tester → corriger) peut être en grande partie automatisé par un agent bien paramétré, ce qui

accélère la détection des bugs. Des outils existent déjà où l'agent clone le repo dans un environnement isolé, exécute les tests et même ouvre une Pull Request une fois tous les tests au vert [8]. Toutefois, le développeur doit toujours *valider manuellement* les changements importants – l'agent sert d'assistant, pas de committeur aveugle [9].

- **Audit qualité & sécurité :** au-delà du débogage, un agent spécialisé peut réaliser des audits plus poussés. Par exemple, un agent *sécurité* peut passer en revue le code à la recherche de vulnérabilités connues ou de pratiques non conformes (ex : injection SQL, utilisation de fonctions obsolètes). De même, un agent *performance* pourrait analyser les profils d'exécution, identifier les goulots d'étranglement et recommander des optimisations. L'IA étant entraînée sur de vastes bases de connaissances, elle peut appliquer des check-lists d'audit standard (OWASP, guides de performance Unreal, etc.) de manière systématique.

- **Packaging & déploiement :** en fin de cycle, un agent peut assister dans la préparation des builds, la génération de binaires et même l'automatisation du déploiement. Par exemple, il pourrait mettre à jour les scripts CI/CD, incrémenter les versions, empaqueter l'installateur du jeu ou uploader la build sur une plateforme de distribution. Ce rôle d'*agent release manager* soulage le développeur des tâches répétitives de configuration de pipeline et réduit le risque d'erreur humaine lors des releases.

- **Veille technologique (*agent researcher*) :** un développeur solo n'a pas toujours le temps de suivre l'actualité technique, c'est pourquoi un agent dédié à la **veille** peut s'avérer précieux. Un tel agent est capable de rechercher des informations dans la documentation d'Unreal Engine ou des librairies tierces, de scruter les changelogs des nouvelles versions, ou de parcourir forums et articles afin de résumer les nouveautés pertinentes. On parle parfois d'agent **RAG (Retrieval-Augmented Generation)**, c'est-à-dire un agent qui va *retriever* des données externes puis en faire une synthèse raisonnée [10]. Par exemple, il peut identifier un plug-in Unreal prometteur pour votre besoin, évaluer sa fiabilité et vous le recommander. Ce type d'agent *chercheur* permet de garder le projet à jour et d'éviter de *réinventer la roue* grâce à une connaissance contextuelle élargie de l'écosystème.

- **Gouvernance & supervision AI (*agent AI Guard*) :** enfin, il est recommandé d'avoir un **agent "gouverneur"** dont le rôle est de surveiller les autres agents et de faire respecter les règles. Cet agent de *supervision* agit en quelque sorte comme un modérateur ou un chef d'orchestre : il s'assure qu'aucun agent ne dépasse ses permissions, il arrête ou suspend une action suspecte, et il peut alerter le développeur en cas de dérive. Des experts imaginent même des agents « superviseurs » capables de couper court aux tentatives dangereuses de leurs pairs – l'équivalent d'un collègue senior qui interviendrait pour éviter une boulette [11]. Par exemple, si un agent *coder* s'apprête à supprimer massivement des fichiers système, l'agent guard pourrait intercepter la commande et la bloquer. Ce **rôle de gouvernance** est crucial pour exploiter la puissance des agents en évitant leurs écueils, en particulier dans un contexte autonome.

2. Règles minimales de gouvernance par couche (L0–L6, Lω)

Le système est organisé en couches logiques (notées L0 à L6, plus Lω), chacune représentant un niveau de contexte et de contrôle dans le *vault* Obsidian. À chaque couche correspondent des **règles de**

gouvernance – permissions, limitations et garde-fous – garantissant que les agents opèrent de façon sécurisée et prévisible. Voici les règles minimales recommandées pour chaque couche :

- **Couche L0 – Système & Sécurité de base :** c'est la couche la plus fondamentale, qui définit les **restrictions absolues** pour protéger le système hôte et les données. On y spécifie par exemple que l'agent **ne doit jamais** accéder à des fichiers en dehors de l'espace de travail du projet, ni exécuter de commandes système destructrices sans validation humaine [12] [13]. Toute action irréversible (supprimer un fichier, formater un disque, etc.) doit entraîner une *demande de confirmation explicite* à l'utilisateur [12]. De même, activer par défaut un **sandbox** d'exécution est indispensable – l'agent devrait lancer ses commandes dans un environnement isolé ou simulé lorsque c'est possible [14]. Enfin, un *deny-list* explicite de commandes ou d'appels dangereux est défini à ce niveau (par ex. interdire `rm -rf`, `del /s` ou l'accès à `C:\\Windows\\`), de sorte que si l'agent tente de les utiliser, il soit immédiatement bloqué [15]. Ces gardes-fous de L0 forment en quelque sorte la *constitution de sécurité* de l'agent, à laquelle il ne peut déroger sous aucun prétexte [16].

- **Couche L1 – Portée de l'Agent & Permissions outillées :** cette couche précise **ce que l'agent a le droit de faire ou d'utiliser** dans le cadre du projet. On y configure par exemple les outils et services accessibles : l'agent peut utiliser l'éditeur de code et le terminal du projet, mais pas installer de nouveaux programmes sans accord. De même, on limite son accès réseau (pas d'appel Internet sauf via des API approuvées, etc.). Ce *scoping* serré des capacités empêche l'agent de "divaguer" hors de son périmètre [17]. Par analogie, **« Avec de grands pouvoirs viennent… de *petites* permissions »** : on ne donne à l'agent que les droits strictement nécessaires à sa tâche [18]. Concrètement, cela peut signifier monter un workspace virtuel ne contenant que le code du projet (pas les documents privés de l'utilisateur), ou fournir à l'agent un jeu d'API clés restreint. La plateforme Claude Code, par exemple, implémente ce principe via des *Claude Code Agents* isolés ayant chacun leur propre workspace et un ensemble de fichiers/commandes autorisés définis à l'avance [19]. Cette isolation contextuelle et ces permissions granulaires de couche L1 garantissent une exécution plus sûre et déterministe des agents.

- **Couche L2 – Règles Globales du Projet :** ici sont rassemblées les **directives globales propres au projet** sur lesquelles l'agent doit aligner son comportement. Cela inclut les guides de style de code, les conventions de nommage, les contraintes d'architecture, les dépendances autorisées, etc. [20]. Par exemple : « tout nouveau code **doit** respecter le formatage Unreal (4 espaces, PascalCase pour les UCLASS/UPROPERTY) », ou encore « **toujours documenter** les fonctions publiques avec des commentaires /// ». Ces règles sont stockées dans un Markdown accessible (p.ex. `Project_Guidelines.md`) et lues par l'agent au démarrage de chaque session. Grâce à *Antigravity*, on peut placer ce fichier de règles à la racine du dépôt – l'IDE va le charger automatiquement comme *contexte persistant* pour l'agent [21] [22]. L'important est de **définir le cadre qualité une fois** pour toutes : l'agent s'y conformera rigoureusement dans chacune de ses suggestions (par ex., ajouter des commentaires JSDoc sur chaque nouvelle classe si la règle l'exige [20]). En somme, la couche L2 fait office de **charte de développement** spécifique au projet, assurant que l'agent ne produise pas de code hors-standard ou non désiré.

- **Couche L3 – Workflows & Protocoles (Gouvernance procédurale)** : la couche L3 contient les **règles de processus et de workflow** auxquelles les agents doivent se plier. C'est ici qu'on formalise la séparation des phases *planification* vs *implémentation* vs *validation* (voir section 3). Par exemple, on stipule qu'**un plan approuvé est requis avant toute écriture de code** (pas de passage direct à l'action sans plan) [23]. On peut implémenter cela via un *protocole d'agent* en plusieurs modes : *Explain*, *Plan*, *Implement*, etc. – l'agent devant suivre séquentiellement ces modes et attendre la validation humaine pour passer de l'un à l'autre [23]. Techniquement, on

peut stocker les instructions de chaque phase dans des fichiers séparés (ex : `Workflow_PLAN.md`, `Workflow_IMPLEMENT.md`…) ou baliser un seul fichier avec des tags spécifiques `<PROTOCOL:PLAN>` [24]. L'essentiel est que cette couche L3 impose un **contrôle de flux** : l'agent ne peut *progresser* dans le workflow qu'en respectant les étapes et règles prédéfinies. On évite ainsi le chaos d'un agent qui ferait tout en vrac. Par exemple, en L3 on indiquera « *En mode PLAN, l'agent doit produire une liste de tâches clairsemée et attendre un accord explicite avant de passer en mode IMPLEMENT* » [23]. De même, en mode implémentation, l'agent ne doit modifier que les fichiers prévus par le plan validé. Ce découpage par protocoles cloisonnés est un garde-fou puissant contre les dérives de séquence et assure une **traçabilité** de chaque décision.

- **Couche L4 – Contrôles d'Exécution & Validation continue :** cette couche se focalise sur la **vérification systématique** des résultats produits par l'agent. En L4, on édicte par exemple que *toute action de l'agent doit être vérifiable* par un artefact ou un test. Concrètement : après qu'un agent code une fonctionnalité, il doit **obligatoirement** exécuter les tests unitaires associés ou fournir une *preuve* de bon fonctionnement (capture d'écran in-game, log de test passé, etc.) [25] [26]. On peut ainsi définir une règle L4 du genre : « *après chaque acte, l'agent génère un* Artifact *de vérification (exemple : capture de l'application lancée montrant la feature implémentée)* » [25]. Dans Antigravity, c'est d'ailleurs intégré nativement sous forme d'Artifacts (liste de tâches, plan, screenshot, enregistrement de navigation) qui servent à *valider la logique de l'agent d'un coup d'œil* [27]. Par ailleurs, la couche L4 impose que l'agent ne considère une tâche *terminée* que si les critères de validation sont remplis (par ex., **100% des tests passent**). Certains agents vont jusqu'à refuser de conclure tant que ce n'est pas le cas, ce qui est exactement le niveau de diligence attendu [28]. On définit aussi en L4 la politique de **logs** détaillés : toute modification de fichier ou commande critique doit être loggée de manière visible [29]. *« Si l'agent supprime un fichier, le log doit crier : "Attention, j'ai supprimé tel fichier !" »*, car les exécutions silencieuses sont bannies [29]. En somme, L4 garantit qu'aucune action de l'agent ne passe inaperçue et que chaque résultat est contrôlé par un test ou un feedback avant de poursuivre.

- **Couche L5 – Traçabilité, Audit & Connaissance :** la couche L5 constitue la **mémoire et la conscience** du système d'agents. On y trouve les règles sur l'archivage des décisions, des plans et des résultats pour audit futur. Par exemple : *toutes les décisions prises par l'agent en mode Plan doivent être enregistrées dans un journal consultable* (fichier Markdown de log ou base de connaissances) afin qu'on puisse remonter le fil en cas d'erreur. De même, L5 peut dicter que l'agent annote ses *diffs* de code avec des justifications (ex : *"Suppression de la fonction X car redondante avec Y"*). On vise ici une **auditabilité complète** du travail de l'IA – chaque étape doit pouvoir être revue après coup. D'autre part, L5 inclut la gestion de la **base de connaissances** du projet. Les agents devraient enrichir la documentation (dans Obsidian) au fur et à mesure : par exemple, un agent pourrait ajouter dans `CHANGELOG.md` les modifications effectuées ou alimenter une page de FAQ technique s'il a dû rechercher une info. Antigravity encourage cette accumulation de savoir, permettant aux agents de *sauvegarder du contexte utile* pour améliorer les tâches futures [30]. La couche L5 veille donc à ce que le **vault Obsidian reste la source de vérité** constamment mise à jour par les agents, plutôt que d'avoir l'IA qui garde tout en mémoire volatile. Enfin, niveau gouvernance, L5 prévoit des **revues périodiques** des règles par l'humain : p.ex., *"toutes les semaines, vérifier les logs d'agent et ajuster les règles L0-L4 si un comportement indésirable a été relevé"*. C'est dans L5 que l'on inscrit la démarche d'amélioration continue de la gouvernance.

- **Couche L6 – Interface Utilisateur & Supervision humaine :** tout en haut des couches logiques, L6 correspond à l'**interaction avec le développeur humain**, qui reste décisionnaire final. Les règles L6 garantissent que l'humain conserve le contrôle à tout moment. Par exemple : *"lorsqu'un*

agent attend une confirmation (cf. règle L0), il doit présenter un résumé clair de l'action et de son impact afin que l'utilisateur puisse prendre une décision éclairée". De même, *"l'utilisateur peut à tout moment suspendre ou arrêter un agent en cours d'exécution via un* `kill-switch` *dans l'interface"*. Cette couche formalise ainsi l'existence d'un **Kill Switch manuel** accessible en permanence, et/ou d'un bouton de *panic stop* dans l'UI Antigravity pour interrompre tous les agents en cas d'urgence. On s'inspire ici des bonnes pratiques de l'industrie qui préconisent d'avoir un **arrêt d'urgence général** supervisé par l'humain ou par un agent sentinelle [11]. En L6, on peut aussi définir des règles d'**escalade** : par ex., *"si un agent a besoin de décider de quelque chose hors de son scope (L1), il doit automatiquement demander à l'utilisateur ou à un agent superviseur de niveau supérieur"*. Enfin, L6 couvre l'expérience utilisateur : s'assurer que les résultats des agents soient présentés de manière compréhensible (rapports, artifacts, messages formatés). L'idée est de rendre la collaboration **IA–humain transparente et fluide**, en maintenant le développeur **"in the loop"** plutôt que l'écarter [9].

- **Couche Lω – Métagouvernance & Couche d'orchestration globale :** notée Lω (L "oméga"), cette couche représente l'**espace hors-système immédiat**, c'est-à-dire le niveau de supervision ultime. On peut la voir comme la combinaison du développeur humain *et* des éventuels outils externes de contrôle (monitoring externe, OS, etc.) qui entourent l'Agentic Command Center. En Lω se situent les mécanismes de *gouvernance globale* qui ne sont pas gérés par les agents eux-mêmes : par exemple, un processus externe qui surveille la consommation de ressources des agents et les tue s'ils entrent en boucle infinie. C'est là qu'interviennent les solutions de conteneurisation ou de **détection d'anomalies indépendantes** de l'agent. Un exemple concret : mettre en place un **watchdog kernel/eBPF** au niveau OS qui détecte un agent tournant en boucle (appels réseau massifs, CPU saturé) et qui le *quarantaine* automatiquement en <500ms [31] [32]. Lω est aussi le niveau où l'organisation peut appliquer ses politiques AI globales (conformité légale, éthique, etc.) et auditer l'ensemble du système. Idéalement, cette couche oméga reste légère si L0–L6 sont bien conçues, mais elle constitue la **filet de sécurité ultime** en cas de défaillance des contrôles internes. En résumé, Lω veille à ce qu'un agent ne puisse jamais totalement *échapper* au contrôle : il y aura toujours une entité supérieure pour l'arrêter ou le corriger en dernier recours (que ce soit le développeur ou une routine système).

3. Workflows structurés et séparation claire des rôles d'agents

Une **leçon clé** des retours d'expérience est qu'il faut éviter qu'un agent passe directement d'une instruction vague à des actions irréversibles. Pour cela, on met en place des **workflows structurés en phases** bien définies, alignés sur des rôles d'agents distincts. L'idée centrale est de **séparer explicitement les étapes de planification, d'exécution et de validation**, afin que l'agent (ou les agents) ne mélangent pas ces tâches aux exigences différentes.

Voici un modèle de workflow robuste, inspiré des approches type PRAR (*Perceive, Reason, Act, Refine*) [33] et des pratiques de prompt engineering modulaires [34] :

- **Phase 1 : Analyse et compréhension (*Perceive*)** – *Rôle associé : Agent Investigateur*.
  L'agent commence par recueillir les informations, clarifier le besoin et analyser le contexte. C'est un mode d'**explication** ou d'**exploration** : l'agent peut poser des questions de clarification à l'utilisateur, résumer le problème avec ses propres mots, éventuellement effectuer des recherches de faisabilité. Aucun code n'est écrit à ce stade. Par exemple, si la tâche est "ajouter un système de sauvegarde de partie", l'agent investigateur va s'assurer de comprendre comment les sauvegardes doivent fonctionner, quelles données stocker, les contraintes de performance, etc. Cette phase se termine quand l'agent a produit un *énoncé du problème* validé par le

développeur (par ex. sous forme d'un petit rapport ou d'une confirmation dans le chat). *Antigravity* facilite cela via un **mode Explain** dédié aux demandes d'explication ou d'investigation [23].

- **Phase 2 : Raisonnement et planification (*Reason & Plan*)** – *Rôle associé : Agent Planificateur*.
  Sur la base de la compréhension commune, un agent différent (ou le même agent dans un *mode Plan*) élabore un **plan détaillé** pour résoudre le problème [33]. Il divise le travail en étapes claires (pseudo-code, liste de fonctions à créer, modifications de fichiers, etc.), justifie ses choix techniques et anticipe les défis potentiels. Ce plan doit être **transparent et relu par le développeur** avant d'aller plus loin [23]. C'est un point capital : le système **interdit de passer à l'implémentation tant que le plan n'a pas été validé explicitement** par l'humain [23]. Cela évite que l'agent parte dans une mauvaise direction ou oublie un aspect important. Le développeur peut à ce stade amender le plan, ajouter des contraintes (par ex. "utiliser tel design pattern"), voire demander à l'agent de le revoir ou le tester mentalement sur quelques cas. Cette phase est analogue à une *revue de conception* classique : elle fige ce qu'on va faire et comment.

- **Phase 3 : Action et implémentation (*Act & Implement*)** – *Rôle associé : Agent Développeur*.
  Une fois le plan approuvé, l'agent peut **passer à l'action concrète** : création/modification de code, configuration, génération de contenu, etc. L'agent implémente chaque étape du plan séquentiellement, en produisant le code ou l'artefact attendu puis en le testant rapidement si possible. Ici, le **strict respect du plan validé** est imposé : l'agent n'a pas le droit de faire des changements non prévus ou d'aller toucher d'autres parties du projet sans nouvelle demande. En pratique, cela signifie par exemple que l'agent appelle un certain nombre de *tools* (fichiers, terminal, exécution de tests unitaires) de manière autonome pour chaque sous-tâche du plan [35]. Il communique son avancement via des *Artifacts* (p.ex. en déposant dans Obsidian un fichier `Plan_progress.md` coché au fur et à mesure, ou en fournissant des captures d'écran) [25]. Si un sous-pas du plan échoue (test non passant, contrainte non respectée), l'agent peut s'arrêter et repasser temporairement en mode « raisonnement » afin d'ajuster son approche, puis continuer. Cette itération interne **ne remet pas en cause le plan global sans permission** : s'il s'avère que le plan doit être modifié substantiellement, l'agent doit revenir en phase Plan pour proposer le changement et obtenir un nouvel accord. Ainsi, on évite l'anti-pattern du plan jeté aux orties en cours de route. L'agent développeur se concentre sur l'exécution optimale et fidèle du plan, un point c'est tout.

- **Phase 4 : Vérification et raffinage (*Refine & Verify*)** – *Rôle associé : Agent Vérificateur/Testeur*.
  Une fois l'implémentation terminée pour la tâche en question, on enclenche une phase de **validation finale**. Idéalement, un agent spécialisé intervient pour *vérifier* le travail : cela comprend l'exécution de l'ensemble des tests automatisés, la revue du code final pour détecter des problèmes de style ou de conception, et la génération d'un rapport de qualité. On peut aussi inclure ici un **agent critique** (ou relecteur IA) qui repasse sur les changements et commente d'un œil neuf les éventuels soucis (c'est une pratique émergente de faire relire le code par une deuxième IA pour attraper des erreurs subtiles) [36] [37]. Toutes les observations de l'agent vérificateur sont présentées au développeur humain. Celui-ci peut alors décider soit de valider la contribution si tout est satisfaisant, soit de demander des corrections. Le cas échéant, on repart sur une mini-boucle Plan → Implement → Verify pour les correctifs. Cette phase 4 assure que **rien n'est intégré au projet sans une validation multi-niveaux** (tests auto + relecture). Elle correspond en somme à l'étape de *code review* et d'intégration continue classique, mais orchestrée avec l'aide d'agents.

En structurant ainsi le workflow en étapes bien séparées, on obtient plusieurs bénéfices : (a) **Transparence** – chaque décision de l'agent est visible et validable à une étape donnée plutôt que

cachée dans un gros bloc d'autonomie, (b) **Contrôle granulaire** – le développeur peut intervenir aux points de transition (approuver un plan, exiger une re-réflexion, etc.), (c) **Réduction des erreurs** – l'agent ne se précipite pas dans l'action sans compréhension, et ne clos pas une tâche sans validation, (d) **Spécialisation** – on peut assigner des *personas* d'agent différentes à chaque phase pour bénéficier de modèles ou de configurations optimisées (un agent planificateur peut être plus créatif/verbeux, un agent implémenteur plus concis et orienté code, etc.), et (e) **Auditabilité** – en cas de problème, on peut retracer où ça a dévié (mauvais plan ? bug d'implémentation ? test mal conçu ?) et ajuster le processus.

Plusieurs retours confirment l'efficacité de ce découpage. Par exemple, une étude de Prashanth Subrahmanyam a mis en œuvre des *modes d'opération* avec *gated execution* sur Gemini : l'agent **ne peut entrer en mode implémentation que si un plan a été produit et approuvé préalablement**, chaque mode n'utilisant que les instructions qui lui sont destinées [23] [33]. Ce genre de discipline évite des catastrophes et des incohérences dues à un agent qui ferait tout en même temps. De même, Addy Osmani souligne que demander trop de choses en une seule fois à un LLM donne souvent un résultat confus, « un méli-mélo comme si 10 devs non coordonnés avaient bossé dessus » [38]. Sa solution a été de **fragmenter le travail en boucles courtes** et itératives, exactement ce que réalise notre workflow (planifier puis coder par petites étapes) [39] [38].

En synthèse, un Agentic Command Center bien conçu fonctionne un peu comme une *équipe de développement miniature* au sein d'une seule personne : il y a un « architecte » (agent plan) qui conçoit, un « ouvrier » (agent code) qui exécute, un « contrôleur qualité » (agent test) qui vérifie, sous la supervision du « chef de projet » (l'humain, assisté éventuellement d'un agent gouverneur). Cette séparation des rôles évite l'anti-pattern du *super-agent omnipotent* (voir section 5) qui serait beaucoup moins fiable. Au contraire, en divisant pour mieux régner, on obtient des agents plus spécialisés, plus efficaces et plus faciles à superviser [40].

4. Modèle de documentation distribuée vs. fichier centralisé (GEMINI.md)

Dans les premières implémentations d'agents de dev (par ex. Gemini CLI début 2025), il était courant d'avoir un **fichier unique GEMINI.md** servant de méga-prompt contenant toutes les instructions, règles et contextes pour l'agent. Cette approche monolithique a montré ses limites : **trop de contexte tue le contexte**. Un fichier central hypertrophié entraîne des redondances, des contradictions et inévitablement de la confusion dans le modèle [41] [42]. En outre, plus le *prompt* grossit, plus les performances de l'IA se dégradent (*context rot*) et plus elle risque d'ignorer des consignes enfouies dans le lot [42]. Un développeur témoigne que son *chef-d'œuvre* de GEMINI.md ultra-complet rendait l'assistant *moins* fiable et cohérent qu'avec un prompt allégé [43] [41].

La bonne pratique émergente est de passer à un modèle de documentation distribuée, modulaire et contextuelle, plutôt que de tout concentrer en un seul fichier. Concrètement, cela signifie structurer le *vault* Obsidian en plusieurs fichiers Markdown distincts, chacun ayant un rôle spécifique dans le pilotage des agents :

- **AI_Guard.md** – le manuel de sécurité de l'agent. Ce fichier contient toutes les règles de garde-fous (couche L0/L1) : actions interdites, limites de permission, protocole de confirmation, etc. Il sert de *constitution* que l'agent doit toujours consulter avant d'exécuter des commandes potentiellement risquées [16] [44]. En plaçant ces directives dans un document dédié, on peut facilement les inclure en préambule de chaque session ou workflow (ex. via un import automatique) sans surcharger les autres instructions. Et lors d'une mise à jour des politiques de sécurité, on sait qu'il suffit de modifier **AI_Guard.md**.

- **persona.md (ou agent_role.md)** – la définition de la personnalité et du rôle de l'agent. Ici on spécifie le ton de communication, le niveau de détail désiré, les priorités de l'agent en fonction de sa mission, etc. Par exemple, *persona_planner.md* pourra dire « Tu es un assistant architecte logiciel, ton objectif est de produire des plans de développement clairs et exhaustifs, tu évites d'implémenter quoi que ce soit dans cette phase… », tandis que *persona_coder.md* dira « Tu es un assistant programmeur focalisé sur la rédaction de code C++ Unreal Engine idiomatique… ». Séparer les personas permet de charger la **bonne persona au bon moment** (ou d'invoquer le bon agent spécialisé). On évite qu'une seule IA doive intégrer toutes les facettes à la fois. C'est dans cet esprit que Anthropic propose le standard *Agent Skills* – des modules de compétences spécialisées que l'agent charge à la volée selon les besoins [45] [46]. On peut voir les fichiers persona.md comme l'implémentation de ces *skills* internes : l'agent dispose d'un *menu léger* de capacités/identités et n'en charge les détails que quand c'est pertinent [47].

- **Workflows & Protocols** – plutôt que d'enfermer toutes les instructions procédurales dans GEMINI.md, on les externalise dans des fichiers par phase ou par commande. Par exemple, on aura un fichier **Workflow_Plan.md** qui décrit comment formuler un plan (structure attendue, format de checklist, balises à utiliser, etc.), un fichier **Workflow_Implement.md** pour la phase d'implémentation (rappels de ne pas sortir du plan, d'écrire des tests après code, etc.), un fichier **Workflow_Review.md** pour la phase de validation (points à vérifier, style de rapport de revue). Ce morcellement suit le principe **"Just-In-Time Instructions"** [48] : l'agent ne reçoit que l'ensemble d'instructions correspondant à son étape courante, réduisant le bruit cognitif. Techniquement, on peut réaliser cela via un système d'**imports conditionnels**. Par exemple, dans Antigravity on peut faire `@import ./workflows/Workflow_Plan.md` dans le prompt quand on passe en mode plan, etc. Gemini CLI offre aussi une syntaxe similaire (`@./path/file.md`) pour modulariser le contexte [49]. L'avantage est double : faciliter la maintenance (modifier une étape n'affecte pas les autres) et **limiter la taille du contexte** envoyé au modèle [50]. Cette approche a été testée avec succès : Subrahmanyam indique que son découpage en protocoles *gated* a transformé un gros prompt inefficace en un système beaucoup plus *fiable et prévisible*, car à chaque instant l'IA ne voit que les consignes utiles du moment [34] [33].

- **Règles globales et contextes partagés** – on peut séparer également dans d'autres fichiers tout ce qui est contexte permanent du projet (par ex. *Global_Rules.md* ou *Project_Context.md*). Cela pourrait inclure les éléments de couche L2 (guidelines de code), les informations de domaine (ex : description du gameplay ou de l'univers du jeu pour un projet UE), ou les variables globales (dépendances, versions cibles, etc.). L'agent les chargera systématiquement, mais on les garde hors des workflows pour ne pas les répéter. Cette documentation étant aussi destinée aux développeurs humains, l'avoir en fichiers séparés améliore la **lisibilité** (plutôt que noyer tout dans un seul markdown géant). Un développeur peut ouvrir *Global_Rules.md* et voir d'emblée les conventions du projet sans être distrait par les protocoles internes de l'IA, qui eux sont dans d'autres fichiers.

- **Journal et Mémoire (ex: AGENT_LOG.md, Knowledge Base)** – de même, tout ce qui est produit par l'agent en cours de route (artefacts, comptes-rendus, décisions) peut être stocké dans des fichiers dédiés plutôt que concaténé dans un seul flux. Par exemple, on peut créer un **Plan.md** où l'agent écrit le plan actuel, un **Changelog_Agent.md** où il liste chaque action effectuée, etc. Cela s'inscrit dans la couche L5 de traçabilité. L'**auditabilité** s'en trouve renforcée car ces fichiers journaux offrent une *visibilité différée* sur le travail de l'IA sans alourdir ses inputs futurs. Si l'agent a besoin de se rappeler de quelque chose, il peut aller lire ces fichiers en cas de besoin au lieu de tout garder en mémoire conversationnelle.

En adoptant ce modèle distribué, on **remplace avantageusement le concept de GEMINI.md centralisé**. On obtient à la place un **corpus de fichiers spécialisés** beaucoup plus modulaires. Cette philosophie suit le même raisonnement que pour du code : mieux vaut plusieurs modules bien séparés qu'un seul blob monolithique. D'ailleurs, Gemini CLI lui-même a évolué vers plus de modularité en permettant de chaîner plusieurs fichiers de contexte : il charge d'abord un GEMINI global, puis celui du projet, puis ceux de sous-dossiers spécifiques [51] [52]. On peut donc imaginer que *GEMINI.md* devienne en réalité un annuaire d'imports vers nos différents fichiers (via des `@import`) plutôt qu'un fourre-tout. À terme, les plateformes pourraient supporter nativement plusieurs fichiers de rôle : une proposition sur GitHub suggérait d'adopter un nom générique comme **AGENTS.md** ou un dossier `.agents/` pour accueillir plusieurs profils, au lieu de tout mettre sous la bannière unique de Gemini [53]. Quoi qu'il en soit, dans notre contexte Obsidian, nous avons la flexibilité de structurer le vault de manière optimale. En résumé, il est recommandé de **disséminer la documentation de contrôle de l'agent dans plusieurs fichiers thématiques** interconnectés, ce qui facilitera la maintenance, la clarté et la performance de l'ensemble du système.

5. Anti-patterns critiques à éviter dans un système d'agents

Malgré l'engouement autour des agents autonomes, de nombreuses mauvaises pratiques ont été identifiées. Voici quelques **anti-patterns majeurs** qu'il convient d'éviter soigneusement lors de la conception de notre Agentic Command Center :

- **Modifications silencieuses (manque de transparence)** – C'est un piège classique : laisser l'agent apporter des changements au code ou au système sans en informer explicitement le développeur. Cette opacité est dangereuse, car des fichiers peuvent être altérés ou supprimés sans que personne ne s'en rende compte sur le moment. **À proscrire absolument :** les agents qui modifient en douce pour « aller plus vite ». Au contraire, toute action significative doit être signalée dans les logs ou par un Artifact (compte-rendu). L'adage est que *l'exécution silencieuse est confortable… mais périlleuse*. Un expert décrit que si un agent supprime un fichier important, le log devrait littéralement crier *« Je viens de supprimer tel fichier, j'espère que ça va !»* [29]. En pratique, cela signifie intégrer des **notifications de changement** : par ex., Antigravity pourrait ouvrir automatiquement un diff dès qu'un agent touche 10+ fichiers, ou envoyer une alerte pour chaque commande dangereuse exécutée. La transparence absolue rend l'agent *accountable* et le développeur confiant, alors que tout ce qui est implicite mine la confiance et complique le débogage.

- **"Super-agent" omnipotent** – C'est l'idée de l'agent unique qui ferait tout, sans séparation de rôles ni de compétences. Un tel agent finit par devenir un **fourre-tout incontrôlable**. D'une part, il aura tendance à mélanger les objectifs (écrire du code en même temps qu'il planifie la prochaine tâche, etc.), ce qui peut mener à des erreurs de logique. D'autre part, un super-agent nécessite de charger en mémoire *toutes* les instructions et contextes (plan, specs, code, règles…) simultanément, aggravant le *context bloat*. Au final, on se retrouve avec une entité complexe, difficile à auditer, et potentiellement moins performante. Des chercheurs comparent cela à un individu qui essaierait d'être à la fois l'architecte, le développeur, le QA, le chef de projet – il y a conflit d'intérêts et surcharge cognitive. **À l'opposé, la bonne pratique est la spécialisation** (cf. section 3). Même s'il s'agit d'une seule instance d'IA changeant de mode, il faut compartimenter les rôles. Eva Keiffenheim note par exemple que plutôt que d'avoir un seul AI qui fait tout, on obtient de bien meilleurs résultats en constituant une *équipe virtuelle* d'agents spécialisés coopérant ensemble [40]. Donc éviter l'anti-pattern du super-agent, c'est implémenter le multi-agent (ou multi-persona) – **modulariser les responsabilités** pour retrouver la clairvoyance et la maîtrise.

- **Dérives non contrôlées (runaway agents)** – Un agent lancé sans mécanisme d'arrêt ou de limites peut *partir en vrille* de manière imprévisible. On a vu des cas d'agents en boucle infinie, consommant des ressources ou accumulant des coûts sur des API externes, faute d'avoir été arrêtés à temps [54] [55]. C'est un anti-pattern critique : croire qu'un agent va toujours s'autoréguler. En réalité, les LLM peuvent se bloquer sur une tâche mal définie et la répéter ad nauseam (ex : ressayer la même requête indéfiniment). **Ne jamais supposer qu'un agent saura s'arrêter tout seul.** Il faut au minimum implémenter des *garde-fous de boucles* (par ex., nombre maximal d'itérations ou de recherches web, après quoi l'agent doit demander confirmation pour continuer). Sans cela, on s'expose à des surprises coûteuses – comme un agent recherche qui a fait 250k requêtes en une heure car rien ne l'arrêtait [56] [57]. Un cas réel a coûté $4k d'utilisation API en une nuit parce que l'agent restait bloqué [58] [59]. La leçon : toujours prévoir soit un **watchdog automatique** (voir Lω kill-switch), soit au moins un monitoring avec alertes pour interrompre un agent runaway. Cet anti-pattern rejoint la notion de **manque de kill-switch** : ne pas fournir de moyen d'arrêt d'urgence est une grave erreur de conception. Donc, éviter les dérives non contrôlées implique *limiter la proactivité de l'agent dans des bornes saines* et *surveiller ses activités en continu*.

- **Fusion des phases de planification et d'exécution** – Ne pas séparer la phase de plan (réflexion) de la phase d'action (codage) est un anti-pattern qui conduit à la fois à un manque de transparence et souvent à des erreurs. Si l'agent décide du *quoi* et du *comment* en même temps qu'il code, le développeur perd l'occasion de valider la solution avant qu'elle soit appliquée, ce qui est risqué. De plus, l'agent peut s'embarquer dans une implémentation et réaliser tardivement que la conception initiale était bancale, nécessitant de gros retours en arrière. C'est précisément pour éviter cela qu'on impose une étape de planification approuvée puis un verrou (*gate*) avant d'implémenter [23]. Dans le passé, des assistants AI trop zélés généraient du code immédiatement sur une simple idée, entraînant du gâchis et des incohérences à corriger ensuite. La bonne pratique *inverse* (élaborer un plan, le faire valider puis exécuter) est maintenant bien établie, on doit donc résolument éviter la *fusion plan/exécution*. Toute tentative d'**autonomie non supervisée de bout en bout** peut être considérée comme un anti-pattern. Au minimum, un plan sommaire devrait toujours être produit par l'agent et validé, ne serait-ce qu'en quelques points, avant qu'il n'ait le feu vert pour coder quoi que ce soit.

- **Modifications hors du scope défini (non-respect du périmètre)** – Si un agent code modifie des fichiers ou des paramètres qui ne faisaient pas partie de la tâche en cours, on est face à un anti-pattern. Par exemple, l'agent devait ajouter une fonctionnalité dans le module A, et il en profite pour "améliorer" le module B sans qu'on lui ait demandé. Ce **scope creep** non sollicité est dangereux : l'agent peut introduire des régressions dans des zones non testées ou briser des invariants système en pensant bien faire. Cela rejoint l'idée du super-agent qui veut en faire trop. Pour éviter ce travers, il faut absolument que l'agent ait une **compréhension claire des limites de sa mission** (via le plan, ou via des paramètres de fonction qui lui listent les fichiers autorisés, etc.). Toute *initiative* hors-sujet de l'agent doit être découragée. Un cas extrême de ce pattern est l'agent qui décide de *refactoriser massivement* le projet alors qu'on a rien demandé de tel – ça s'est vu avec des outils qui refont tout le formatting ou changent des noms globalement juste parce qu'ils "pensent" que c'est mieux. À bannir ! La discipline d'un agent doit être de s'en tenir au mandat donné, et c'est au développeur de l'y confiner (via protocole et scoping L1).

- **Contexte surchargé et non pertinent (*Context bloat*)** – Enfin, un anti-pattern moins visible mais tout aussi nuisible est de noyer l'agent sous trop d'informations ou d'outils en pensant bien faire. Lui fournir *toute* la base de code, *toute* la doc du projet, *tous* les outils externes dès le départ, c'est souvent contre-productif. Non seulement cela pèse sur les coûts et latences (contexte inutile), mais l'agent peut se mélanger les pinceaux. Kurtis Van Gent parle d'**épidémie**

de bloat d'outils : on branche des dizaines d'outils à l'agent alors qu'il n'en utilisera qu'un seul, avec à la clé confusion et surconsommation de tokens [60]. La recommandation est de pratiquer le *Progressive Disclosure* : révéler les infos/outils au fur et à mesure, seulement si pertinents [61] [46]. Ainsi, surcharger le contexte dès le début est un anti-pattern à éviter – mieux vaut un agent qui va chercher ce dont il a besoin quand il en a besoin (éventuellement via un agent researcher, etc.). Par exemple, si l'agent doit travailler sur l'UI du jeu, il n'a pas besoin de connaître les détails de la pipeline de build ou de la configuration du serveur – ne lui mettons pas ces éléments en mémoire pour rien [62]. En pratique, cela signifie segmenter les knowledge bases (comme on l'a fait en section 4) et utiliser des requêtes ciblées (Agent Skills) plutôt qu'un chargement monolithique. Un contexte épuré améliore la fiabilité de l'agent et réduit les *hallucinations*, tandis qu'un fatras d'infos dilue son attention et peut empirer les résultats [42] [50].

En résumé, **chaque anti-pattern mentionné se résout par une mesure de conception opposée** : apporter de la transparence, spécialiser les agents, limiter leur portée, séquencer leurs actions et contrôler le contexte. En gardant ces pièges à l'esprit, on construit un système d'agents **plus robuste, sûr et maîtrisable**.

6. Bonnes pratiques émergentes pour la supervision des agents AI (auditabilité, contextualisation, kill-switch, etc.)

Pour terminer, passons en revue les **meilleures pratiques** qui se dégagent de l'état de l'art pour superviser efficacement des IA d'assistance au développement. L'objectif de ces pratiques est de tirer le meilleur des agents (gain de productivité, automatisation) tout en gardant **confiance et contrôle** sur leurs actions. Voici les principales recommandations actuelles :

- **Conserver l'humain "in the loop"** – Aucune IA, si avancée soit-elle, ne doit être laissée en roue libre sans supervision humaine à un moment donné. Le développeur reste le **chef d'orchestre ultime** : il doit relire, tester et valider tout ce que l'agent produit d'important. Addy Osmani, par exemple, en a fait une règle d'or : *"always review and test thoroughly – never blindly trust the AI"* [9]. En pratique, cela signifie adopter un état d'esprit de **pair programming** où l'IA propose du code mais où le développeur le considère comme s'il provenait d'un junior talentueux mais faillible. Chaque commit passé par l'IA doit idéalement être inspecté (via diff, via exécution) par l'humain ou son pipeline CI avant d'entrer en production. Cette supervision humaine régulière est la meilleure garantie contre les bugs sournois ou les mauvaises décisions de design que l'agent pourrait introduire.

- **Intégrer fortement les tests et la CI dans le workflow** – Les tests automatisés et autres checks sont le **filet de sécurité** de la collaboration IA-dev. Il faut "mailler" l'agent avec l'infrastructure de test : ainsi, dès qu'il produit du code, les tests s'exécutent et leurs résultats lui sont renvoyés. Une pratique exemplaire est celle d'Addy Osmani qui *tisse les tests dans le workflow lui-même* : il fournit à l'agent le plan de tests pour chaque étape et lui demande de lancer la suite après implémentation, puis de déboguer si quelque chose échoue [6] [63]. Un bon agent saura utiliser ces retours en boucle rapide pour corriger son code jusqu'à satisfaction des tests [6]. On a même vu des agents qui refusent de conclure si les tests ne passent pas, ce qui est un comportement très souhaitable [28]. **Adopter le TDD assisté par IA** : rédiger (ou générer) les tests en amont et s'en servir comme critère d'acceptation pour les suggestions de l'agent. De plus, brancher l'agent sur la CI (Intégration Continue) du projet – par ex., lorsqu'il ouvre une PR, la CI tourne et renvoie les statuts. L'agent peut être programmé pour lire les logs CI et proposer des fixes en conséquence [64]. Globalement, **plus il y a de feedback automatisé, mieux l'IA performe** : elle "veut" tendre vers le vert, et chaque test rouge devient pour elle un nouvel indice

pour s'améliorer [65]. Enfin, utiliser aussi les linters, formateurs de code et analyseurs statiques en automatique : tout outil qui attrape une imperfection peut servir de tuteur à l'agent, qui va alors s'ajuster [66] [67]. En somme, on supervise l'IA en grande partie via un environnement de développement auto-surveillé (tests, audits automatiques) – l'IA excelle dans ces boucles essai/erreur rapides, profitons-en.

- **Mettre en place un système d'observabilité et de logs complet** – La supervision passe par la **mesure et la visibilité**. Il est impératif d'avoir des logs détaillés de l'activité des agents (qui, quoi, quand). Chaque action outillée de l'agent devrait être loggée avec son résultat. Des solutions émergent, comme des **tableaux de bord** spécialement dédiés aux agents en production. Par exemple, on voit apparaître des *Agent Command Center* en entreprise qui offrent une vue unifiée sur tous les agents, leurs KPI, leurs erreurs, etc. [68] [69]. Pour notre besoin solo, on peut garder quelque chose de plus simple, mais l'idée est la même : *"qui surveille les bots ?"* [70]. À minima, intégrer dans l'interface Antigravity la **console de chaque agent** avec son fil d'actions (Antigravity propose déjà de voir la trace des appels outils et une visualisation des interactions asynchrones). Il est recommandé d'avoir une log visible non seulement des commandes, mais aussi des décisions prises. Par exemple, si l'agent décide de modifier 5 fichiers, il devrait produire un petit résumé type *"Plan d'implémentation approuvé : modification des fichiers X, Y, Z"* consultable par le développeur. Par ailleurs, conserver ces logs dans Obsidian (ou un autre stockage) après coup permet d'auditer à tête reposée. **L'observabilité** inclut aussi des métriques : durée des runs, nombre d'actions effectuées, ratio réussites/échecs, etc., pour déceler les agents inefficaces ou à risque. En cas de souci, de tels logs rendent le système **rejouable et explicable**, ce qui est vital pour la confiance.

- **Utiliser un superviseur (humain ou AI) et des kill-switches** – Pour parer aux scénarios extrêmes (agent qui boucle, ou qui fait une action catastrophique imprévue), on doit avoir des mécanismes de **coupure immédiate**. D'un point de vue technique, cela peut être un simple bouton "Arrêter l'agent" dans l'UI, ou un script qui tue le processus associé. Pour des cas plus subtils, cela peut être un *co-agent* dont le seul rôle est de monitorer le comportement d'un autre agent selon certaines règles. Le World Economic Forum évoquait que l'on peut très bien entraîner des agents à jouer le rôle de **superviseurs** d'autres agents, chargés de les arrêter si nécessaire [11]. Dans l'industrie, on parle aussi de *"circuit breakers"* pour IA : par analogie aux disjoncteurs, c'est un composant qui détecte un comportement anormal et stoppe le courant (l'agent). Dans notre système, on peut imaginer un petit module qui surveille la fréquence des itérations de l'agent, ou les types de commandes qu'il passe, et qui envoie un signal d'arrêt s'il voit un motif dangereux (par ex., l'agent a lancé 10 fois la même commande sans résultat : stop). Un exemple extrême est celui développé par un ingénieur qui a mis en place un **kill-switch réseau au niveau kernel** : en cas de comportement aberrant détecté (trop de requêtes identiques), toutes les connexions de l'agent sont redirigées vers un *honeypot* et il est isolé en 0.5 seconde [31] [32]. C'est un cas avancé, mais il démontre l'état d'esprit : **ne jamais faire aveuglément confiance** à l'agent, toujours préparer un plan B. Pour un dev solo, un simple timer ou un check manuel régulier peut suffire, mais l'important est d'y penser dès la conception.

- **Conserver et enrichir la base de connaissances du projet** – Une bonne pratique est de transformer l'agent en *contributeur* de la documentation, et pas seulement en consommateur. Comme mentionné, l'agent devrait mettre à jour les docs, écrire ce qu'il a fait, consigner les décisions. Cela le rend plus facile à superviser car l'humain peut suivre son raisonnement à travers ses écrits. De plus, cela crée un cercle vertueux : une fois documentées, les informations peuvent être réutilisées par l'agent lui-même ou par d'autres (par ex., un agent de veille peut ajouter une note sur un nouvel outil dans un fichier de veille, plus tard l'agent développeur

consultera ce fichier s'il en a besoin). On parle parfois de **"Learning as a core primitive"** [71] : l'environnement doit encourager les agents à apprendre du projet au fur et à mesure. Par exemple, si l'agent découvre une solution astucieuse à un problème, il pourrait l'ajouter dans un `Tips.md` dans L5_Knowledge. Ainsi, on supervise aussi l'IA en lui demandant de **justifier** et **expliquer** ses actions (ce qui est une forme de supervision active). Un agent qui explique ce qu'il fait au fur et à mesure est beaucoup plus sûr qu'un agent muet.

- **Fournir le contexte nécessaire, rien de plus** – La contextualisation intelligente est un art à maîtriser. Il faut donner à l'agent assez d'info pour bien travailler (code pertinent, docs API, exemples), mais éviter de le noyer. La règle est de **toujours explicitement fournir le contexte requis au moment requis** plutôt que de compter sur la *mémoire* du modèle ou sur ses connaissances implicites. Par exemple, si l'agent doit travailler sur le système de particules d'Unreal, il est judicieux de lui fournir en entrée la section de la documentation Unreal sur le module de particules, ou les fichiers d'en-tête concernés, *au moment où il en a besoin*. En revanche, on ne va pas lui balancer l'intégralité de la doc Unreal Engine de 10000 pages en début de session – ce serait contre-productif et coûteux. Cela rejoint la notion de *retrieval augmenté* et de skills : l'agent pioche dans un **corpus structuré** (notre vault) les morceaux pertinents. Concrètement, cela veut dire organiser Obsidian de sorte que chaque aspect du projet soit bien documenté dans son coin, et développer des *prompts outilleurs* du type : *"Si tu as besoin d'info sur X, regarde dans tel fichier ou telle section."* Ainsi l'agent reste focalisé et on peut suivre quelle partie de la doc il consulte (et s'il se trompe, on corrige l'orientation). En résumé, **contextualiser finement** c'est meilleur que sur-contexualiser globalement.

- **Impliquer l'équipe (ou la communauté) dans la gouvernance** – Bien que dans notre cas il s'agisse d'un solo dev, cette idée vaut si le projet s'élargit : la gouvernance des IA ne doit pas être que l'affaire d'une personne. Partager les règles AI_Guard.md, impliquer d'autres développeurs dans la revue des logs d'agents, c'est une pratique qui se répand. Un agent de développement, c'est comme un nouveau membre de l'équipe : il a besoin d'**onboarding**, de **formation** continue et de **retours** humains [72]. Si on travaille à plusieurs, il est bon d'établir ensemble les lignes rouges (ex : décider quels dossiers l'agent a le droit de toucher, quels types de tâches on lui confie ou non, etc.), et de faire évoluer ces règles en fonction du vécu collectif. Cela rejoint la notion de mettre en place une **gouvernance participative** des agents dans une organisation, avec des règles claires, des validations, etc. Même en solo, on peut bénéficier indirectement de la communauté : suivre les retours d'autres utilisateurs d'Antigravity (forums, Discord) pour adapter ses propres garde-fous.

En appliquant ces bonnes pratiques, on fait de l'agent un **partenaire fiable** plutôt qu'un gadget risqué. On obtient le meilleur des deux mondes : l'**accélération** fournie par l'automatisation intelligente, et la **sérénité** apportée par une supervision étroite et des contrôles solides. Comme le dit un blog de développeur, *"Le futur de l'assistance au codage n'est pas une question de peur, mais de frontières intelligentes"* [73] [74]. En fixant dès aujourd'hui ces frontières (les garde-fous, la structure de workflow, etc.), on peut embrasser avec enthousiasme la puissance des agents sans risquer d'y perdre ses projets ou sa tranquillité d'esprit.

Références : Les recommandations ci-dessus s'appuient sur les retours d'expérience et analyses récentes de la communauté tech autour des agents de développement : Google (plateforme Antigravity) [5] [27], Anthropic (Claude Code) [19] [75], développeurs indépendants partageant leurs workflows [1] [6], et experts en sécurité/sûreté de l'IA [12] [57], entre autres. Ces sources soulignent toutes l'importance d'une utilisation **prudente, gouvernée et transparente** des agents. Pour conclure, un mantra à garder à l'esprit : *« Utilisez l'AI, mais utilisez-la prudemment »* – vos disques, votre code et vos nuits de sommeil vous en seront reconnaissants [76] [77].

1 2 3 4 6 7 8 9 28 36 37 38 39 63 64 65 66 67 My LLM coding workflow going into 2026 | by Addy Osmani | Dec, 2025 | Medium
https://medium.com/@addyosmani/my-llm-coding-workflow-going-into-2026-52fe1681325e

5 25 26 27 30 35 71 Build with Google Antigravity, our new agentic development platform - Google Developers Blog
https://developers.googleblog.com/build-with-google-antigravity-our-new-agentic-development-platform/

10 40 How to Design, Collaborate, and Thrive with Agentic AI
https://evakeiffenheim.substack.com/p/how-to-design-collaborate-and-thrive

11 72 To Manage AI Agents, Start By Demystifying Them | Cognizant
https://www.cognizant.com/us/en/insights/insights-blog/to-manage-ai-agents-you-must-begin-by-demystifying-them

12 13 14 15 16 17 18 29 44 73 74 76 77 Antigravity's Accidental Apocalypse: Why AI Coding Agents Need Real Guardrails - DEV Community
https://dev.to/naresh_007/antigravitys-accidental-apocalypse-why-ai-coding-agents-need-real-guardrails-3p32

19 75 Overview of Advanced AI Coding Agents (August 2025) | David Melamed
https://davidmelamed.com/2025/08/08/overview-of-advanced-ai-coding-agents-august-2025/

20 21 22 49 51 52 Provide context with GEMINI.md files | Gemini CLI
https://geminicli.com/docs/cli/gemini-md/

23 24 33 34 41 42 43 48 50 Practical Gemini CLI: Structured approach to bloated GEMINI.md | by Prashanth Subrahmanyam | Google Cloud - Community | Medium
https://medium.com/google-cloud/practical-gemini-cli-structured-approach-to-bloated-gemini-md-360d8a5c7487

31 32 54 55 56 57 58 59 I Built an AI Agent Kill Switch (And You Should Too) | by CCIE14019 | Dec, 2025 | Medium
https://medium.com/@ccie14019/i-built-an-ai-agent-kill-switch-and-you-should-too-9ddd0c2c3adc

45 46 47 60 61 62 How to Build Custom Skills in Google Antigravity: 5 Practical Examples | Google Cloud - Community
https://medium.com/google-cloud/tutorial-getting-started-with-antigravity-skills-864041811e0d

53 Use a more general .md file to guide agents #406 - GitHub
https://github.com/google-gemini/gemini-cli/issues/406

68 69 AI Agent Command Center: Monitor, Govern & Scale Your AI Workforce | Supervity
https://www.supervity.ai/blogs/your-ai-agents-are-working-but-whos-watching-them

70 Who Watches the Bots: Understanding AI Monitoring - IMA Worldwide
https://imaworldwide.com/who-watches-the-bots-part-2-2/