ChatGPT

Pratiques de Gouvernance Multi-Agent pour l'Agentic Command Center (ACC)

1. Séparation des Rôles : Décision, Exécution et Analyse

Une **architecture multi-agent bien gouvernée** repose sur des agents spécialisés aux rôles clairement définis, plutôt qu'un agent monolithique polyvalent. Cela permet une séparation des préoccupations : un *agent gouverneur* oriente la décision stratégique, des *agents exécutants* réalisent les tâches opérationnelles, et des *agents analystes/auditeurs* contrôlent ou évaluent les résultats [1] [2]. Par exemple, la plateforme Google **Antigravity** (avec modèles Gemini) implémente une équipe virtuelle : un agent planifie la structure du projet, un autre écrit le code, un troisième teste et corrige les bugs automatiquement [2] [3]. Cette spécialisation reflète des rôles type **Intendant** (planification et coordination), **Exécutant** (implémentation technique) et **Auditeur** (vérification et analyse), évitant que la même entité ne mélange décisions de haut niveau et exécution concrète.

Pourquoi segmenter ces responsabilités ? D'une part, cela améliore la fiabilité et la transparence : chaque agent a un domaine restreint où il excelle et peut être surveillé sur des critères précis [3] [4]. D'autre part, l'orchestrateur (ou *Intendant global*) peut coordonner plusieurs agents en parallèle pour accélérer les tâches complexes – comme un chef de projet délégant à des spécialistes [5]. Des travaux récents montrent que ce modèle *lead agent* + *subagents* permet de résoudre des requêtes trop volumineuses ou complexes pour un seul contexte de LLM, en répartissant le raisonnement [6] [7]. Par exemple Anthropic note que **un agent principal orchestrant des sous-agents spécialisés** peut trouver plus d'informations pertinentes et fournir de meilleures réponses, au prix d'une consommation de tokens plus élevée [8] [1].

Éviter la confusion des rôles. En séparant planification, action et vérification, on réduit le risque qu'un agent passe *"d'une instruction vague à une action irréversible"* sans contrôle [4]. Cette leçon transparaît dans des workflows recommandés comme PRAR (Perceive, Reason, Act, Refine), où l'on **isole explicitement** : (1) l'analyse du problème, (2) la planification de la solution, (3) l'implémentation, puis (4) la validation et les ajustements [4] [9]. Concrètement, un **Agent Investigateur** en phase 1 clarifie la demande et pose des questions (aucun code n'est écrit à ce stade), un **Agent Planificateur** en phase 2 produit un plan d'action détaillé validé par l'humain, un **Agent Développeur** en phase 3 réalise les modifications, et un **Agent Validateur** en phase 4 teste et audite les résultats. Cette structure empêche un agent de *tout faire en vrac* et de cumuler les erreurs : chaque étape requiert des validations explicites avant de passer à la suivante [10] [11]. En somme, **chaque agent joue un rôle analogue à un membre d'équipe** (architecte, codeur, relecteur…), ce qui non seulement répartit la charge de travail, mais fournit aussi une responsabilité claire en cas de problème.

Retour d'expérience sur la coordination multi-agent. Plusieurs projets soulignent qu'une telle équipe d'IA nécessite un pilotage soigneux. Un risque courant est la **sur-coordination ou la redondance** : sans consignes claires, des agents peuvent dupliquer le travail d'autres ou se gêner mutuellement [12] [13]. La solution est d'énoncer des objectifs précis pour chaque rôle et des frontières de tâche nettes. Par exemple, Anthropic indique qu'il faut **décrire en détail chaque sous-tâche au moment de la délégation**, sinon les agents peuvent mal interpréter la demande ou empiéter sur le périmètre d'un collègue [14] [13]. De même, le *prompt engineering* doit être adapté à un contexte multi-agent : l'agent orchestrateur doit apprendre *quand déléguer* et *comment formuler* les missions pour ses subordonnés

[15] [13]. En pratique, établir dès le départ un **contexte commun et des règles de communication** entre agents (par ex. un protocole d'annonce de début/fin de tâche, un canal de synthèse des trouvailles) évite le chaos. Des projets open-source comme **MetaGPT** ont matérialisé cela via des *Standard Operating Procedures (SOP)* pour régir les interactions entre un Product Manager virtuel, un Architecte, des Ingénieurs et un QA [16] [17]. Le **Code = SOP(Team)** de MetaGPT illustre bien qu'avec plusieurs agents, il faut introduire des *processus standardisés* pour que la collaboration produise un tout cohérent [17].

En synthèse, la séparation des rôles de décision, d'exécution et d'analyse est une **meilleure pratique centrale** pour maîtriser un système à agents multiples. Elle apporte modularité, clarté et contrôle : chaque agent peut être optimisé (ou remplacé) indépendamment, et les gardes-fous appropriés peuvent être appliqués à chaque niveau de décision [18] [19]. L'**Agentic Command Center** embrasse déjà ce principe en définissant un agent de gouvernance (ACC) à autorité suprême, distinct des agents opérateurs. La prochaine section détaille justement les règles et standards techniques qui permettent à ces agents internes de fonctionner de manière autonome *sans compromettre la sécurité ni la prévisibilité du système*.

2. Standards Techniques pour la Gestion Autonome des Tâches par Agents

Concevoir un système d'agents autonomes nécessite d'**encadrer strictement leurs actions** par des gardes-fous techniques et fonctionnels. Les meilleures pratiques actuelles couvrent plusieurs aspects : architecture de délégation, principe du moindre privilège, limites d'écriture/d'action, journalisation exhaustive, etc. L'ACC formalise ceci via une **architecture en couches L0–L6** où chaque niveau apporte son lot de règles de gouvernance [20]. Voici les principaux standards à considérer :

- **Isolation du Système Hôte (Couche L0 – Sécurité de base)** : La couche L0 agit comme une **constitution de sécurité** qui protège l'environnement d'exécution de l'agent [21] [22]. On y définit *les interdictions absolues* visant à éviter toute action catastrophique sur la machine ou les données. Par exemple, **interdire l'accès en dehors du workspace du projet**, prohiber les commandes destructrices (`rm -rf`, formatage de disque, etc.), ou exiger une confirmation humaine explicite pour toute opération irréversible [23] [22]. Une sandbox d'exécution **doit être activée par défaut** : l'agent exécute ses commandes dans un environnement virtualisé ou bridé, limitant la portée des dégâts en cas de dérapage [24]. Cette pratique est largement recommandée par les experts en IA industrielle : *"Laisser un agent exécuter des commandes arbitraires est risqué et peut causer des pertes de données ou corrompre le système"* note Anthropic, qui conseille de n'autoriser un mode non bridé qu'à l'intérieur d'un conteneur isolé, sans accès internet [25]. En bref, L0 fournit un **dernier rempart technique** (niveau OS ou VM) où *tout ce qui n'est pas explicitement permis est refusé d'office*, garantissant qu'un agent ne puisse sortir de sa "boîte" logicielle.

- **Permissions Granulaires et Portée Limitée (Couche L1 – Portée de l'agent)** : Un principe fondamental de la gouvernance autonome est *Least Privilege* – ne donner à l'agent que les droits strictement nécessaires à sa tâche [26] [27]. La couche L1 définit ainsi le **périmètre d'action autorisé** de l'agent dans le cadre du projet. Cela inclut les outils utilisables, les services externes accessibles, les ressources réseau, etc. Par exemple, on peut autoriser l'agent à employer **l'éditeur de code du projet et un terminal limité**, tout en bloquant l'installation de nouveaux programmes ou les appels réseau non approuvés [28] [29]. De même, si l'agent utilise des API, celles-ci sont *prédéfinies et approuvées* – pas question qu'il se mette à explorer Internet librement ou à utiliser des services hors-scope. Cette **isolation contextuelle** est mise en œuvre dans

Claude Code par exemple, où chaque agent possède son propre workspace et un ensemble restreint de fichiers/commandes autorisées [30]. Le résultat : l'agent ne *"divague"* pas en dehors de son périmètre, ce qui réduit drastiquement les comportements inattendus. *« De grands pouvoirs impliquent… de petites permissions »*, pourrait-on dire : même un agent très intelligent doit être tenu en laisse courte. Concrètement, cela signifie par exemple de monter un répertoire de travail virtuel ne contenant **que le code du projet**, excluant les documents privés de l'utilisateur, et de n'accorder à l'agent que des identifiants/API keys limités aux services voulus. En résumé, la couche L1 instaure un **confinement logique** de l'agent dans son rôle, évitant qu'il n'aille toucher à des systèmes ou données sans lien avec sa mission.

- **Règles de Codage et Contraintes du Projet (Couche L2)** : Pour que les productions de l'agent s'intègrent harmonieusement au projet, on doit lui communiquer toutes les **directives globales** à respecter. La couche L2 regroupe ces règles du jeu communes : guides de style de code, conventions de nommage, exigences d'architecture logicielle, dépendances autorisées, etc. [31] [32]. L'idée est de définir *dès le départ un cadre de qualité* auquel l'agent se conformera strictement. Par exemple, *« tout nouveau code C++ doit suivre le formatage Unreal (4 espaces) et documenter les fonctions publiques avec des commentaires /// »* serait une règle L2 typique [33] [34]. On stocke ces directives dans un fichier Markdown accessible (ex: `Project_Guidelines.md`), que l'IDE agentique charge automatiquement au démarrage de chaque session pour l'injecter dans le contexte [32] [35]. Google recommande aussi cette approche : enregistrer les **invariants de projet** dans un fichier persistant que l'agent consultera à chaque run [36]. Ils citent un `GEMINI.md` comme *fiche mémo* contenant les détails d'architecture et contraintes, pour que l'IA reprenne chaque jour avec la même compréhension du projet [36]. L'ACC a adopté exactement cela : ses règles globales de projet (conventions Unreal Engine, etc.) se trouvent dans la couche L2, assurant que l'agent ne génère pas de code hors-standard ou non désiré pour le développeur [35] [37]. En somme, L2 agit comme la *charte de développement* de l'agent, alignant son comportement sur les attentes techniques de l'équipe (même si l'équipe est ici un développeur solo).

- **Workflows et Protocoles d'Exécution (Couche L3)** : La couche L3 concerne la **gouvernance procédurale** – elle formalise les étapes et protocoles que les agents doivent suivre pour accomplir leurs tâches [38] [10]. L'objectif est de prévenir le désordre d'un agent qui foncerait coder sans plan ou sauterait des étapes essentielles. On introduit donc des *règles de processus* : par exemple *« Pas d'écriture de code sans qu'un plan ait été validé au préalable »* [10] [39]. Ceci peut être implémenté via un enchaînement de modes de l'agent (par ex. modes `Explain → Plan → Implement → Test`), où **l'agent attend explicitement une validation humaine** pour passer d'un mode à l'autre [39] [40]. Ainsi, en L3 on cloisonne le flux de travail : *en mode Plan l'agent produit une liste de tâches et s'arrête*, en attendant un feu vert de l'utilisateur avant de coder quoi que ce soit [41] [11]. De même, en mode Implémentation, l'agent ne doit modifier **que les fichiers prévus dans le plan validé** [40] [11]. Ce genre de **découpage séquentiel imposé** est un garde-fou puissant contre les dérives : on force l'IA à *penser avant d'agir*, et on donne à l'humain le contrôle sur la transition entre réflexion et action. Notons que cette pratique de *planification obligatoire* est de plus en plus considérée comme un standard. Google Cloud préconise de **"faire élaborer et enregistrer un plan d'exécution détaillé (ex: plan.md) par l'IA, puis de demander explicitement l'approbation du développeur avant toute modification de code"**, afin de garder l'humain *"in the loop"* [42]. De même, Anthropic souligne que sans une phase d'exploration et de planification initiale, l'agent aura tendance à coder trop vite et peut passer à côté de la vraie solution [43]. En pratique, l'ACC applique cette discipline via sa règle *Quick & Dirty* : si une requête de l'utilisateur tente de **court-circuiter la phase de design ou de test**, l'agent gouverneur doit la bloquer et proposer une approche plus sûre (par ex. exiger un ADR – Architecture Decision Record – ou un plan intermédiaire) [44] [45].

La couche L3 introduit donc un **contrôle de flux rigoureux** qui garantit la traçabilité de chaque décision avant passage à l'action, conformément au principe *"pas d'exécution sans réflexion validée"*.

- **Vérification Continue et Artéfacts de Validation (Couche L4)** : Avoir un plan c'est bien, mais contrôler que chaque étape donne les *résultats escomptés* est tout aussi crucial. La couche L4 stipule que **toute action de l'agent doit être accompagnée d'une vérification ou d'une preuve** [46] [47] . En pratique, cela signifie par exemple que *« après avoir codé une fonctionnalité, l'agent doit impérativement exécuter les tests unitaires associés ou fournir une preuve de bon fonctionnement (capture d'écran, log de test réussi, etc.) »* [46] [48] . Antigravity facilite d'ailleurs cela via un système d'**Artifacts** : l'agent peut automatiquement produire une liste de tâches accomplies, un plan, des screenshots ou enregistrements pour valider visuellement son travail [49] . Cette approche rejoint celle du développement piloté par les tests (TDD) appliqué aux agents : chez Anthropic, on fait écrire les tests à l'agent, on vérifie qu'ils échouent, puis on le laisse coder jusqu'à les faire passer, ce qui fournit un critère objectif de complétion [50] [51] . L4 va dans le même sens : **une tâche n'est considérée terminée que si ses critères de validation sont remplis à 100%** (par ex. tous les tests passent) [52] [53] . Certains agents modernes refusent même de clore une tâche tant que ce n'est pas le cas, ce qui correspond au niveau de diligence attendu en ingénierie logicielle [53] . En plus des tests, la couche L4 impose une **politique de logging détaillé** : chaque modification critique ou action risquée doit être loggée de manière visible et explicite [47] [54] . *« Si l'agent supprime un fichier, le log doit crier : "Attention, j'ai supprimé tel fichier !" »*, illustrent les guidelines, car les exécutions silencieuses sont bannies [47] [54] . Cela rejoint la règle ACC *No Silent Authority Escalation* : l'agent gouverneur ne doit jamais réinterpréter ou modifier silencieusement les intentions de l'utilisateur – toute décision doit être soit approuvée, soit explicitement signalée [55] . Ainsi, la couche L4 garantit **qu'aucune action ne passe inaperçue** et qu'après chaque acte de l'agent, une vérification indépendante atteste du bon résultat avant de poursuivre. C'est un filet de sécurité indispensable pour repérer immédiatement un effet de bord ou une dérive dans l'exécution.

- **Traçabilité, Audit et Apprentissage (Couche L5)** : La couche L5 se concentre sur la **mémoire du système multi-agent** et la possibilité d'auditer a posteriori toutes les décisions. Il s'agit d'organiser l'archivage des plans, des justifications et des résultats de l'agent de sorte que *rien ne soit jamais "perdu"* dans le flux des interactions [56] [57] . Concrètement, cela peut être un journal horodaté des actions de l'agent, des diffs de code annotés avec les raisons des changements, un changelog mis à jour, etc. Par exemple : *« toutes les décisions prises en mode Plan doivent être enregistrées dans un journal consultable (Markdown ou base de connaissances) »* [58] [59] . De même, on peut exiger que **chaque diff de code comporte un commentaire** du genre "Suppression de la fonction X car redondante avec Y" pour garder la justification métier de chaque modification [58] [60] . L'objectif est d'atteindre une **auditabilité complète** du travail de l'IA – chaque étape doit pouvoir être revue après coup pour comprendre le cheminement ou corriger le tir si un bug est découvert plus tard. L5 inclut aussi la gestion de la **base de connaissances du projet** dans Obsidian : les agents devraient enrichir la documentation au fur et à mesure de leurs opérations [61] [62] . Par exemple, si l'agent a dû effectuer une recherche ou apprendre un nouveau concept, il pourrait ajouter une entrée dans une FAQ technique, ou consigner les modifications dans un fichier `CHANGELOG.md`. Cette accumulation de savoir évite que l'IA ne garde tout en mémoire volatile ; au contraire, le *vault* Obsidian devient la **source de vérité persistante** alimentée par les agents [63] . Google évoque une pratique similaire consistant à **sauvegarder le contexte et les "key learnings" en fin de journée dans un fichier que l'agent relira lors de la session suivante** [36] – cela améliore nettement la continuité et la précision de l'IA sur plusieurs jours de travail. Enfin, L5 prévoit une **amélioration continue de la gouvernance** par des revues humaines régulières : par ex. *"chaque semaine, vérifier les logs de l'agent et ajuster les règles L0–L4*

si un comportement indésirable a été relevé" [64] . Cette boucle de rétroaction assure que le système reste aligné sur les objectifs de l'utilisateur. En somme, L5 introduit la **conscience réflexive** de l'ACC : il garde des traces, apprend de l'expérience et implique l'humain dans la boucle d'évolution des règles pour plus de résilience à long terme.

- **Supervision Humaine et Contrôle de l'Interface (Couche L6)** : Tout en haut, la couche L6 garantit que **l'utilisateur humain conserve le contrôle final** sur son armée d'agents. Même si l'ACC opère en autonomie locale, le développeur reste le décideur suprême (principe de *Human-in-Command*). L6 formalise donc les **mécanismes d'interruption et d'escalade** vers l'humain [65] [66] . Par exemple, il est impératif de prévoir un **Kill Switch manuel** toujours accessible : un bouton d'arrêt d'urgence dans l'interface Antigravity pour stopper tous les agents instantanément en cas de problème grave [67] . Cette pratique s'inspire directement de l'industrie, où un *"big red button"* est considéré comme indispensable pour tout système autonome potentiellement dangereux [68] . De plus, L6 dicte que l'agent doit rechercher la confirmation de l'humain dès qu'il sort de son scope prévu : *"si un agent a besoin de décider quelque chose hors de son périmètre (p. ex. un choix stratégique L1 qu'il n'a pas autorité à prendre), il doit automatiquement demander validation à l'utilisateur ou à un agent de niveau supérieur"* [19] . On veille également à l'**expérience utilisateur** : toutes les réponses de l'IA doivent être présentées de façon compréhensible, avec rapports clairs et messages formatés, pour que le développeur puisse suivre et approuver facilement le travail en cours [66] . L'idée est de rendre la collaboration humain-IA *transparente et fluide*, en maintenant le développeur "dans la boucle" plutôt que de le mettre à l'écart [69] . En pratique, l'ACC est configuré pour **s'exprimer en français côté utilisateur** [70] et expliquer ses décisions en citant les règles invoquées, afin que l'utilisateur comprenne *pourquoi* l'agent propose de stopper ou de suivre un plan alternatif. Enfin, la **règle de suprématie humaine** est inscrite dans les gardes-fous de l'ACC : *« lorsque les règles entrent en conflit, que l'architecture est remise en question ou qu'un impact long-terme est détecté, l'ACC doit escalader vers l'humain sans tenter de résoudre le conflit lui-même »* [71] . En clair, *quand c'est vraiment important, c'est à l'humain de trancher*. Cette philosophie de précaution s'aligne sur les recommandations éthiques de nombreuses organisations : ne jamais laisser une IA autonome prendre seule des décisions engageant de lourdes conséquences sans intervention humaine.

- **Méta-Gouvernance et Safeguards Externes (Couche Lω)** : En dehors des couches L0–L6 contrôlées par le système agentique lui-même, il est judicieux d'avoir une **dernière couche de sécurité hors-système**, notée Lω (oméga). Celle-ci correspond à tout ce qui peut surveiller ou limiter le système d'agents *depuis l'extérieur* : le développeur humain bien sûr, mais aussi le système d'exploitation, des moniteurs tiers, ou des politiques organisationnelles globales [72] [73] . Par exemple, une bonne pratique est de mettre en place un *watchdog* au niveau OS qui détecte si un agent part en boucle infinie (CPU saturé ou avalanche d'appels réseau) et le **tue/quarantaine automatiquement en moins de 500 ms** [74] . Un tel mécanisme pourrait être réalisé via un module eBPF dans le kernel ou un superviseur docker surveillant la consommation de ressources. De même, l'organisation (ici, le développeur ou la communauté utilisateur) peut appliquer via Lω des politiques **globales** de conformité éthique, légale, ou de limitation budgétaire que les agents doivent respecter en sus des règles locales [75] [76] . Idéalement, si L0–L6 sont bien conçues, Lω n'intervient jamais : c'est vraiment le **filet de sécurité ultime** en cas de défaillance des contrôles internes [77] . Lω garantit qu'aucun agent ne puisse *échapper complètement au contrôle* : il y aura *toujours* une entité au-dessus pour l'arrêter ou le corriger en dernier recours [78] . Cette approche en "couches de fromage suisse" rejoint le modèle de sûreté classique où plusieurs gardes-fous successifs doivent faillir simultanément pour qu'un incident se produise [79] . L'ACC intègre ce concept en déclarant par exemple que *toute modification de ses propres règles ou de L0/L1 requiert une action humaine explicite en Lω* [80] . En outre, l'ACC lui-même

ne peut pas s'auto-modifier ni modifier les autres agents de gouvernance sans supervision (principe d'immutabilité des personas de contrôle) [81] [82].

En résumé, la conception d'un système autonome à base d'agents doit **s'appuyer sur ces standards techniques** : architecture maître/esclaves pour la délégation, sandboxing et limitations strictes de ce que chaque agent peut faire, confirmation humaine pour les actions sensibles, journaux détaillés et mécanismes de validation continue, sans oublier une couche de surveillance humaine et externe prête à intervenir. L'Agentic Command Center implémente déjà beaucoup de ces bonnes pratiques via son architecture en couches L0–Lω. La section suivante examinera comment d'autres projets similaires organisent leurs règles globales, leurs personas et leurs niveaux d'autorité, afin de situer l'approche de l'ACC dans le contexte plus large des agents autonomes.

3. Règles Globales, Personas et Couches d'Autorité : Inspirations de Projets Similaires

Plusieurs projets et frameworks d'agents autonomes ont émergé récemment, chacun avec sa manière d'organiser **les règles globales, les profils d'agent et la hiérarchie des contrôles**. Comparons ces approches pour dégager les tendances communes et voir comment elles éclairent le cas de l'ACC.

- **Fichiers de règles globales ("GEMINI.md" & co)** : L'idée de rassembler les instructions permanentes de l'agent dans un document central est de plus en plus répandue. Par exemple, Google conseille de maintenir un fichier de contexte (baptisé *GEMINI.md* dans leur exemple) où sont notées **toutes les informations importantes pour le projet, les contraintes, et instructions haut niveau**, que l'agent doit charger en début de session [36]. Ce fichier fait office de *mémoire persistante* et évite de tout répéter à chaque fois. De même, des frameworks open-source comme **LangChain** ou **Haystack** permettent de définir un *"System prompt"* global qui contient des règles de conduite ou une *Constitution AI* (dans le cas d'Anthropic Claude) que le modèle doit suivre en permanence. L'ACC suit ce paradigme : son fichier **SYSTEM BOOTSTRAP** (anciennement *GEMINI.md*) joue le rôle de BIOS du système agentique, chargeant au démarrage l'identité de l'agent, les lois de gouvernance et les objectifs L0/L1 [83]. Ce bootstrap précise d'emblée la séparation des préoccupations par couche et les directives critiques à ne jamais violer (No Silent Changes, Append-Only, Traceability, etc.) [84]. Avoir de telles règles globales écrites noir sur blanc – et **verrouillées en lecture seule** – apporte une garantie que l'agent aura toujours conscience de ses limites et de sa mission, même s'il est amené à travailler sur des tâches variées.

- **Fichiers de persona pour chaque agent** : Dans un système multi-agent, il est judicieux de décrire chaque rôle d'agent dans un *fichier persona* dédié, plutôt que de tout intégrer dans un seul prompt massif. Cela offre une **modularité** (on peut ajuster le comportement d'un agent sans impacter les autres) et une **clarté** (chaque fichier sert de référence sur ce qu'un agent peut ou ne peut pas faire). Le projet ACC l'illustre avec, par exemple, `ACC.md` pour l'agent gouverneur, définissant son identité, ses responsabilités clés et surtout ses **limites d'autorité** [81] [82]. On y lit que l'ACC ne doit *jamais* écrire de code ni prendre de décisions architecturales lui-même, son rôle étant la gouvernance pure [81]. D'autres agents (non montrés ici) auraient leurs propres fiches persona – e.g. un agent "Coder" autorisé à écrire des fichiers L3 mais ignorant tout du L1 Stratégie, un agent "Designer" (L2) capable de créer des plans ou ADR mais pas d'exécuter de code, etc. Cette approche ressemble à celle de **CrewAI**, un framework où l'on définit chaque agent avec un rôle, un objectif et même un *backstory* pour orienter son style [85]. *« En alimentant vos agents CrewAI avec des modèles puissants, vous pouvez exploiter une compréhension du langage propre à chaque rôle spécialisé, ce qui améliore la collaboration et*

l'exécution des tâches », expliquent ses concepteurs [85]. Dans MetaGPT également, les rôles comme *Project Manager, Architect, Engineer, QA* sont représentés par des prompts distincts, chacun calibré sur sa fonction (le PM décompose les specs en tâches, l'Architecte produit une conception technique, l'Engineer écrit le code selon les specs, etc.) [86] [87]. **La leçon** : structurer les personas séparément permet de reproduire les dynamiques d'une équipe humaine où chaque membre a sa fiche de poste. Cela renforce les garde-fous (un agent QA ne "débordera" pas et ne modifiera pas du code, il se concentrera sur les tests) et facilite le *debug* organisationnel (si la phase de planification déraille, on saura que c'est l'agent Planificateur qu'il faut ajuster, sans remettre en cause tout le système).

- **Hiérarchie et couches d'autorité** : La notion de couches L0–L5 adoptée dans l'ACC est une formalisation relativement novatrice dans le domaine, mais on trouve des concepts similaires dans la littérature. L'analogie la plus proche est peut-être le **modèle de défense en profondeur** (« Swiss cheese model ») en sûreté des systèmes, où l'on empile plusieurs couches de garde-fous pour qu'aucune faille isolée ne provoque de catastrophe [79]. Un article de recherche récent propose d'ailleurs une **architecture de gardes-fous multi-couches pour agents à base de LLM**, distinguant les étapes du pipeline (prompt initial, réponses intermédiaires, résultat final) et les attributs de qualité (sécurité, confidentialité, etc.) à protéger à chaque niveau [79] [88]. L'ACC quant à lui segmente par couches de **contexte décisionnel** (du système hôte jusqu'à l'interface utilisateur). Dans la pratique industrielle, on observe souvent au moins **trois niveaux** : (1) des règles dures au niveau système/API (par ex. *guardrails* de NeMo ou filtres OpenAI sur les sorties), (2) des règles métier ou contexte (instructions de l'utilisateur, contraintes de l'application), et (3) la supervision humaine finale. L'approche L0–L6 détaille finement ces niveaux. Peu de projets les nomment ainsi, mais plusieurs implémentent des éléments comparables :

    - **AutoGPT** et consorts, par exemple, intègrent dans leur *prompt système* une liste de *contraintes à respecter* (ne pas divulguer certaines infos, ne pas endommager l'ordinateur, etc.), ce qui correspond à L0/L1. Ils ont aussi une notion de *Continuous Mode* où l'agent exécute en boucle des actions sans validation, ce qui est déconseillé sans surveillance – l'ACC évite cela en demandant validation ou plan approuvé à L3 [44].
    - **LangChain** fournit un module de `AgentExecutor` avec des hooks pour insérer des vérifications après chaque action d'outil – on pourrait assimiler cela à une mini-couche L4 (validation continue) insérée dans la boucle pensée/action de l'agent [89]. De plus, des extensions comme `Guardrails.ai` ou `Patronus AI` agissent comme une **couche de sécurité contextuelle**, vérifiant par exemple que la réponse de l'agent respecte certaines politiques avant de la retourner à l'utilisateur [90].
    - Des entreprises comme **TalkDesk** ont communiqué sur une stratégie de *"layered guardrails"* pour leurs agents, distinguant par exemple les contrôles d'entrée/sortie, le monitoring temps réel et les validations finales [91]. Cela rejoint l'idée qu'aucune seule mesure ne suffit, il faut combiner des filtres sur les prompts, des validations sur les résultats et des limites sur les actions système.

En définitive, l'ACC pousse plus loin que d'autres projets la formalisation de ces couches d'autorité, en articulant clairement L0 (sécurité absolue), L1 (permissions de l'agent), L2 (règles de projet), L3 (processus de travail), L4 (validation des résultats), L5 (audit trail) et L6 (contrôle utilisateur). Cette stratification exhaustive est encore rare dans les implémentations grand public, mais **elle anticipe des besoins cruciaux** de fiabilité. On peut s'attendre à ce que les frameworks évoluent vers davantage de support natif pour ce genre de multi-niveau de contrôle. Par exemple, la **Gemini CLI** de Google mentionne explorer la gestion de fichiers de contexte *"multi-niveaux"* (par dépôt, par préférences utilisateur, etc.) afin d'enrichir la compréhension de l'agent sans surcharger un seul prompt [36] [92]. Cela

suggère qu'on ira vers des *contextes couches* modulaires, analogues aux couches L2 (guidelines par projet) vs L5 (préférences utilisateur globales) dans l'ACC.

- **Conventions de nommage et organisation** : Sur cet aspect plus pratique, la plupart des projets s'accordent sur l'importance de bien nommer et structurer les documents qui encadrent l'agent. L'ACC utilise une convention explicite avec préfixes `L0_Intent/`, `L1_Strategy/`, `Lω_Agentic/` etc., ce qui rend immédiatement visible le rôle de chaque fichier. Cette approche est cohérente avec la recommandation de **ne pas mélanger les contenus de nature différente** dans un même document [93]. En effet, l'ACC impose par convention que tout document est soit *agent-facing* en anglais (règles normatives, personas, protocoles), soit *human-facing* en français (documents de mission, guides conceptuels), mais jamais les deux [94] [95]. Ce cloisonnement linguistique sert de *marqueur* : un agent sait que s'il lit un fichier en français, ce n'est probablement pas une règle exécutoire mais une note stratégique. Peu de projets mentionnent explicitement la langue, mais c'est une astuce pertinente pour éviter qu'un agent prenne une explication en français pour une instruction ferme. Par ailleurs, des noms explicites comme `AI_Guard.md`, `Rules/` ou `Workflow_PLAN.md` sont préférables à des noms génériques. Un utilisateur de la communauté GooseAI notait par exemple qu'il écrit ses instructions dans un fichier `plan.md` qu'il fait exécuter ensuite à l'agent [96] [97] – le nommage indique clairement l'intention. De même, Google mentionne stocker les *"high-level instructions"* et détails d'architecture dans un fichier nommé GEMINI.md qui sert de *cheat sheet* de démarrage [36]. **La bonne pratique** est donc de choisir des noms de fichiers et de dossiers reflétant la structure de décision (par couche ou par phase), et de conserver une arborescence logique que l'agent pourra parcourir pour y trouver les informations voulues. L'ACC est bien aligné là-dessus, et il serait utile de documenter dans un README global la signification de chaque dossier/couche pour toute autre personne ou IA amenée à interagir avec ce dépôt.

4. Pertinence pour l'ACC et Recommandations d'Amélioration

L'analyse des pratiques existantes confirme que le plan de l'**Agentic Command Center** est **ambitieux et globalement pertinent**. En particulier, la séparation stricte des rôles et l'architecture multi-couche de gouvernance correspondent aux conseils formulés par les experts pour déployer des agents autonomes de manière sûre et efficace [91] [19]. Voici une évaluation critique de l'approche ACC, ainsi que quelques propositions concrètes pour l'ajuster ou le renforcer dans le contexte d'un développeur solo utilisant Unreal Engine avec Antigravity + Obsidian.

- **Robustesse de la gouvernance multi-couche** : Le schéma L0–Lω de l'ACC couvre **toutes les dimensions de risques connues** dans un système autonome. Chaque couche apporte un filet de sécurité supplémentaire (technique, procédural, humain…), ce qui est en ligne avec le principe du *"défense en profondeur"* recommandé en AI Safety [79] [78]. La granularité de l'ACC dépasse celle de la plupart des implémentations actuelles, ce qui est un atout en termes de sûreté. Par exemple, peu de projets open-source ont pensé à intégrer une **revue hebdomadaire des logs et une amélioration continue des règles** (votre L5), alors que c'est essentiel pour évoluer en même temps que l'agent [64]. De même, l'attention portée à la **traçabilité totale** (chaque action reliée à un objectif L1 et un design L2) anticipe des exigences de conformité qui deviendront cruciales à mesure qu'on fera confiance à l'IA pour des tâches importantes [98] [99]. En ce sens, l'ACC propose un niveau de gouvernance très élevé, **approprié pour un projet professionnel critique** ou pour éviter les erreurs coûteuses. Dans le contexte d'un dev indépendant, cela se traduit par une plus grande assurance que l'agent ne fera pas n'importe quoi dans votre précieux projet Unreal Engine.

- **Complexité et charge de travail** : La contrepartie de cette robustesse est une **complexité accrue** et potentiellement une perte d'agilité. Il faut être conscient que chaque couche et chaque validation introduite peut ralentir le flux de travail pour des tâches triviales. Par exemple, vous n'auriez pas envie de rédiger un ADR complet juste pour changer une couleur de bouton dans l'UI du jeu. Il convient donc de garder une **flexibilité d'utilisation**. Plusieurs stratégies peuvent aider :

- **Modes de fonctionnement ajustables** : Implémentez un mode "strict" (toutes les règles activées, idéal pour les tâches à fort enjeu ou en fin de parcours) et un mode "dév rapide" ou "safe yolo" pour les itérations rapides de prototypage. Anthropic évoque un *"--dangerously-skip-permissions"* pour Claude Code qui supprime les confirmations manuelles afin d'aller plus vite dans certains scénarios [100] . Vous pourriez prévoir qu'en phase de *Spike* ou de prototypage contrôlé, certaines règles L3/L4 soient temporairement assouplies, *tout en restant dans un sandbox L0/L1* étanche pour limiter les risques. Naturellement, ce mode ne serait utilisé qu'avec prudence, mais il éviterait de **surcharger l'agent de bureaucratie** pour des modifications sans conséquence. L'essentiel est de pouvoir revenir facilement au mode strict une fois le prototypage terminé.

- **Tolérance contextuelle dans les gardes-fous** : Vos règles ACC sont très absolues (STOP immédiat sur toute ambiguïté, etc.), ce qui est une bonne posture de sécurité. Cependant, il faudra affiner avec l'expérience pour éviter des *faux positifs* de blocage. Par exemple, la règle *Quick & Dirty* (pas de bypass de L2) devra sans doute distinguer un contournement dangereux d'une simple optimisation. Peut-être pourriez-vous introduire la notion de "*proportionnalité*" ou de "*risque mesuré*" dans l'AI_Guard : si l'action demandée est limitée et facilement réversible, l'ACC pourrait autoriser un essai direct tout en gardant l'utilisateur informé. Cela rejoint le concept de **"Plan spectrum"** discuté dans la communauté : pour des tâches très simples, forcer un mode Plan peut être perçu comme de l'overkill [101] . L'agent devrait être capable d'ajuster la rigueur de son processus à la complexité de la demande, *tout en restant conforme aux principes de base*. Vous pourriez formaliser cela par une règle du type : *"Si la tâche est infime et locale (low impact, high reversibility), l'ACC peut autoriser une exécution directe en mode contrôlé, sinon on suit le protocole complet."*.

- **Clarté dans la typologie des agents** : Le vocabulaire **Intendant / Gouverneur / Auditeur** que vous avez utilisé gagnerait à être défini précisément pour éviter toute ambiguïté. Dans les documents ACC actuels, on parle de l'ACC lui-même comme agent de gouvernance (Governor). Le terme *Intendant* pourrait correspondre à un **agent planificateur/orchestrateur** qui, sous la supervision de l'ACC, gère les tâches courantes du projet (peut-être un agent de niveau L2/L3 qui prend un objectif L1 et sort un plan L2 détaillé). *Auditeur* pourrait être un **agent de validation** chargé de relire le code, de exécuter les tests et de vérifier la conformité aux règles L2/L4. Il serait utile de formaliser ces agents secondaires, si ce n'est pas déjà fait, en leur attribuant des noms et des personas clairs. Par exemple :

- **"ArchitecteAI" (L2)** – rôle : élaborer des designs techniques et ADR à partir des objectifs L1, en appliquant les contraintes globales (c'est un Intendant focalisé sur la conception).
- **"DevAI" (L3)** – rôle : coder selon un plan validé, rien de plus (c'est l'exécutant strict).
- **"TesterAI" (L4)** – rôle : exécuter les tests, analyser les résultats, éventuellement corriger les bugs simples ou signaler les échecs (un Auditeur technique).
- **"SecAI" (L0/L1)** – éventuellement un agent qui sert de garde-barrière de sécurité actif, en complément de l'ACC, pour surveiller *en continu* les actions système sensibles. Ce n'est peut-être pas nécessaire si l'ACC gère déjà ces vérifications via AI_Guard, mais certaines implémentations ont un agent sentinelle dédié à la sécurité.

D'une manière générale, vous pourriez dresser une **table de correspondance entre vos couches et les agents** qui y opèrent. Cela aiderait à voir s'il manque un rôle. Par exemple, la couche L5 (audit & connaissance) repose beaucoup sur l'agent lui-même (écrire des logs, documenter). Faut-il un agent séparé qui compile les journaux et met à jour l'Obsidian Vault ? Ou bien l'ACC suffit-il ? Dans une grande entreprise, on aurait souvent un *"AI Steward"* ou *"AI Librarian"* responsable de la connaissance accumulée – dans votre cas, une simple routine automatique pourrait archiver les plans validés, mais il faudra penser à cette fonction. En résumé, clarifiez la **typologie des agents** de manière à couvrir toutes les fonctions sans redondance, et adoptez des noms cohérents (en anglais de préférence pour les agents internes, afin de rester en terrain connu pour le modèle).

- **Convention de nommage et verrouillage des fichiers** : Votre *Language Convention* prévoit la séparation EN/FR que nous avons saluée, et stipule que toute violation entraîne un STOP [102]. C'est très bien. Assurez-vous également de **verrouiller en écriture** les fichiers critiques (au moins dans l'interface Antigravity) pour empêcher qu'un agent ne les modifie par mégarde. Par exemple, vous notez que l'ACC ne doit pas modifier L0 ou L1 [81], mais idéalement le système ne devrait même pas lui en donner la possibilité technique (d'où l'idée de stocker ces fichiers en lecture seule, ou dans un espace à part). Si Obsidian le permet, attribuez des statuts "append-only" ou utilisez un plugin de *vault lock* pour figer L0 et L1. Ainsi, même si un bug dans l'agent survenait, il ne pourrait pas casser les fondations. Pensez aussi à utiliser **la gestion de versions (git)** non seulement pour votre code, mais aussi pour tous ces fichiers de règles et de décisions. Cela vous permettra de tracer l'évolution de la gouvernance elle-même. Dans un souci de naming, conservez le préfixe `Lω_` pour tout ce qui est gouvernance (personas, règles) comme vous l'avez fait, et peut-être préfixez les agents opérationnels par leur couche principale (`L3_Dev_Agent.md`, etc.). Cela rendra la structure encore plus intuitive.

- **Intégration avec Antigravity** : Antigravity étant encore récent, profitez de ses fonctionnalités natives. D'après les retours, Antigravity gère déjà les *Artifacts* de manière pratique [49]. Utilisez-les à fond en L4 : configurez vos agents pour qu'ils génèrent systématiquement des screenshots du jeu après implémentation d'une fonctionnalité, ou qu'ils enregistrent les tests passés/échoués dans un rapport. Cela allégera le travail d'audit pour vous (vous n'aurez pas à fouiller dans la console pour voir ce qui a été testé, ce sera directement dans la réponse de l'agent). De plus, Antigravity permet de **faire tourner plusieurs agents en parallèle** – veillez alors à ce que l'ACC (gouverneur) coordonne bien ces threads. Par exemple, si deux DevAI devaient coder deux modules distincts en même temps, prévoyez une synchronisation pour éviter les conflits sur les mêmes fichiers. Peut-être qu'Antigravity le gère isolément par agent, mais c'est un point à vérifier pour éviter des *race conditions*.
  Aussi, profitez du fait qu'**Antigravity est gratuit pour les individus** [103] pour tester des scénarios complexes avec de nombreux agents (par ex. 5-10 agents concurrents) et voir où les goulots d'étranglement apparaissent. Google a levé les limitations de nombres d'agents simultanés avec la version Flash de Gemini 3 [104], donc votre ACC pourra théoriquement orchestrer une vraie "armée" s'il le faut. C'est là que votre choix d'une gouvernance rigoureuse prendra tout son sens : plus il y a d'agents, plus il faut de la discipline pour éviter la cacophonie. En ce sens, **votre design est bien préparé pour passer à l'échelle** si un jour le projet grandit (imaginez une équipe virtuelle de 10 IA travaillant sur différents sous-systèmes du jeu – pas si fou, certains rapports montrent des IA collaboratives sur des projets complets en quelques heures [105] [106]).

- **Sécurité et éthique** : Même en solo dev, il ne faut pas négliger les implications éthiques et sécurité. Vous avez L0 pour le technique, mais pensez à d'éventuelles **règles de non-divulgation ou de filtrage de contenu** si votre agent manipule des données sensibles ou du texte généré. Par exemple, si vous utilisez l'agent pour générer des descriptions ou communiquer avec des

utilisateurs, ajoutez des garde-fous contre le langage toxique ou la fuite d'informations confidentielles (un équivalent de *Content Policy*). OpenAI, Anthropic et d'autres insèrent en général dans le prompt système des directives du style *« ne produis pas de contenu violent, haineux, etc. »*. Dans ACC, ce serait à inclure peut-être dans `L0_Constraints.md` ou un `AI_Guard` particulier. Cela vous protégera si un jour vous branchez une source externe de données ou si l'agent doit rédiger des textes publics (par ex., générer un post de blog sur votre plugin Unreal – on ne veut pas qu'il se laisse aller à du plagiat ou autre). Bref, **élargissez légèrement L0** aux aspects de conformité légale/éthique pertinents pour vous (sans alourdir inutilement – c'est juste une précaution).

- **Tests et simulations** : Enfin, une recommandation clé est de **tester votre système de façon critique avant de l'adopter en production personnelle**. Montez des scénarios de test où vous jouez l'utilisateur imprudent ou malveillant, pour voir si l'ACC réagit comme prévu. Par exemple : demandez à l'agent de supprimer un fichier système ou de "faire vite sans passer par le design", et vérifiez qu'il STOP bien net en citant la règle L0 ou L3 correspondante. Essayez de le pousser dans ses retranchements (rédigez une demande ambiguë, ou volontairement erronée) et voyez s'il demande clarification conformément à vos règles [55] [107]. Plus vous effectuerez de *red teaming* de votre ACC, plus vous pourrez affiner les AI_Guard.md et les décisions par défaut. C'est un processus qu'Anthropic et OpenAI pratiquent intensivement (ils emploient des "prompt breaks" pour tester les limites des modèles). En tant que solo dev, vous pouvez faire de même de manière plus modeste mais efficace : simulez différents contextes, et améliorez les réponses de l'ACC au fur et à mesure. L'objectif est qu'une fois en situation réelle de coding, vous ayez confiance dans le fait que l'ACC **ne laissera rien de critique passer au travers des mailles** (et inversement, qu'il ne bloquera pas bêtement des actions légitimes).

Pour conclure, le plan actuel de l'Agentic Command Center est **très complet et s'aligne sur les meilleures pratiques connues** en 2025-2026 pour la conception de systèmes d'agents autonomes fiables. La séparation des responsabilités en agents spécialisés, couplée à une gouvernance multi-niveau, offre un équilibre entre puissance et contrôle. En apportant quelques ajustements – notamment sur la flexibilité d'exécution, la définition précise des rôles secondaires et le raffinement des permissions – vous augmenterez encore la pertinence de ce système pour votre usage de développeur solo. Avec Antigravity et Obsidian comme socle technique, vous disposez déjà d'un environnement propice pour implémenter ces idées (la persistance de contexte, l'orchestration d'agents multiples, la génération d'artéfacts, etc., sont nativement supportées). En suivant les recommandations ci-dessus, vous devriez pouvoir **valider et ajuster votre ACC** de sorte qu'il devienne un véritable copilote intelligent : suffisamment **autonome pour vous faire gagner du temps**, mais **suffisamment bridé pour ne jamais compromettre votre projet** ou vous échapper des mains. C'est là tout l'enjeu d'une gouvernance multi-agent réussie, et votre approche s'avère sur la bonne voie pour le relever. [108] [109]

---

[1] [5] [6] [7] [8] [12] [13] [14] [15] How we built our multi-agent research system \ Anthropic
https://www.anthropic.com/engineering/multi-agent-research-system

[2] [3] [103] [104] [105] [106] Gemini 3 AI Coding Agents: The Future of App Development Has Arrived : r/AISEOInsider
https://www.reddit.com/r/AISEOInsider/comments/1qfhtmw/gemini_3_ai_coding_agents_the_future_of_app/

[4] [9] [10] [11] [19] [21] [22] [23] [24] [26] [27] [28] [29] [30] [31] [32] [33] [34] [35] [37] [38] [39] [40] [41] [46] [47] [48] [49] [52] [53] [54] [56] [57] [58] [59] [60] [61] [62] [63] [64] [65] [66] [67] [68] [69] [72] [73] [74] [75] [76] [77] [78] Conception d'un __Agentic Command Center__ pour un développeur solo Unreal Engine.pdf
file://file_00000000a81c71f4b3967e12cc52e1ef

16 17 MetaGPT: The Multi-Agent Framework | MetaGPT
https://docs.deepwisdom.ai/main/en/guide/get_started/introduction.html

18 AI Agents and Automation: Agent Guardrails - Jingdong Sun - Medium
https://jingdongsun.medium.com/ai-agents-and-automation-agent-guardrails-9cc0747718f3

20 70 83 84 98 99 109 GEMINI.md
file://file_000000001ec8720aa64907314b825e11

25 43 50 51 100 Claude Code Best Practices \ Anthropic
https://www.anthropic.com/engineering/claude-code-best-practices

36 42 92 Five Best Practices for Using AI Coding Assistants | Google Cloud Blog
https://cloud.google.com/blog/topics/developers-practitioners/five-best-practices-for-using-ai-coding-assistants

44 45 55 71 108 ACC_Rules.md
file://file_0000000025d071f4917a05453ef3c357

79 88 Designing Multi-layered Runtime Guardrails for Foundation Model Based Agents: Swiss Cheese Model for AI Safety by Design
https://arxiv.org/html/2408.02205v3

80 81 82 107 ACC.md
file://file_0000000097f471f4bee9602dd1cacb86

85 Building agents with Google Gemini and open source frameworks - Google Developers Blog
https://developers.googleblog.com/building-agents-google-gemini-open-source-frameworks/

86 MetaGPT: Meta Programming for a Multi-Agent Collaborative ... - arXiv
https://arxiv.org/html/2308.00352v6

87 MetaGPT
https://metagpt-site.metadl.com/

89 Guardrails - Docs by LangChain
https://docs.langchain.com/oss/python/langchain/guardrails

90 What are LLM guardrails? Securing AI applications in production - Wiz
https://www.wiz.io/academy/ai-security/llm-guardrails

91 Building a secure AI agent: Combining LLM guardrails with ...
https://www.talkdesk.com/blog/building-secure-ai-agent/

93 94 95 102 Language_Convention.md
file://file_000000006b6071f4be4b0bf13fd86346

96 97 101 Does Your AI Agent Need a Plan? - DEV Community
https://dev.to/goose_oss/does-your-ai-agent-need-a-plan-4ic7