Agent de Recherche Documentaire pour Base de Connaissances Markdown

Projet d'Agentification - Motifs de Conception Agentiques

Angela, Aurélien, Baptiste et Khaled

20 Décembre 2025

Table des matières

1 Présentation de la Problématique et Cadrage

1.1 Contexte et Motivation

La gestion de connaissances personnelles à travers des systèmes de prise de notes comme Obsidian ou Notion s'est généralisée, particulièrement dans les domaines académiques et techniques. Les utilisateurs accumulent des centaines, voire des milliers de notes markdown organisées hiérarchiquement, couvrant des sujets variés allant de l'intelligence artificielle aux projets professionnels. Dans le cadre de ce projet d'agentification, nous avons identifié la recherche documentaire dans ces bases de connaissances comme un cas d'usage pertinent nécessitant une transformation en système agentique.

Le vault cible est notre vault personnel et contient plus de 200 fichiers markdown structurés en dossiers thématiques : `School/Notes/AI/` (cours ML, DNN, NLP, PyTorch, GenAI), `Oncology/` (projets medical imaging sur ovarian, pancreas, uterus), `Reading/` (articles, books, newsletters), et `Cuisine/` (recettes organisées par catégorie). Cette richesse informationnelle hétérogène pose un défi idéal pour démontrer l'apport des motifs de conception agentiques.

1.2 Problématique Centrale

La problématique s'articule autour de trois défis majeurs justifiant une approche agentique :

1.2.1 Inefficacité des approches traditionnelles

Les outils de recherche classiques (grep, recherche full-text) nécessitent une connaissance préalable de la structure du vault et des mots-clés exacts. Pour un utilisateur cherchant "comment fonctionne le backpropagation", un simple grep retournera trop de résultats non pertinents, tandis qu'une recherche manuelle fichier par fichier est chronophage et inefficace.

1.2.2 Surcharge contextuelle des modèles de langage

Les fichiers markdown longs (300-1000 lignes) génèrent 2000-4000 tokens lorsqu'ils sont lus intégralement. Cette surcharge du context window dilue l'information pertinente et dégrade la qualité des réponses. Un agent simple qui lit naïvement les fichiers complets consomme des tokens inutilement et "se perd" dans le contenu, illustrant la nécessité d'une stratégie de retrieval structurée.

1.2.3 Complexité du raisonnement multi-étapes

Les questions réalistes nécessitent plusieurs opérations séquentielles avec dépendances. Par exemple, "Résume mes notes sur les transformers et explique l'attention" requiert : (1) chercher les fichiers pertinents, (2) identifier la section sur l'attention, (3) extraire le contenu, (4) synthétiser. Un système non-agentique échoue à décomposer et orchestrer ces étapes.

1.3 Apport de l'Agentification

L'agentification transforme ce workflow de recherche en un système autonome capable de :

— **Percevoir** l'environnement (structure du vault, contenu des fichiers)
— **Planifier** des séquences d'actions adaptées à la requête
— **Maintenir un état interne** (mémoire des fichiers consultés, contexte conversationnel)
— **Raisonner** sur la qualité de ses réponses et s'auto-corriger
— **Interagir** avec des outils spécialisés de manière autonome

Cette approche contraste avec un simple système de question-réponse qui se contenterait de répondre à des requêtes isolées sans planification ni apprentissage.

1.4 Motifs de Conception Agentiques Intégrés

Conformément aux exigences du projet, notre architecture implémente **quatre motifs de conception agentiques** complémentaires détaillés dans ce rapport :

— Planner-Executor (Pattern Principal)

— Module de Mémoire (Court et Long Terme)

— Réflexion et Critique (Self-Vérification)

— Tool Use Pattern

1.5 Objectifs Techniques et Contraintes

1.5.1 Contraintes techniques

— **Exécution locale CPU** : Pas de GPU disponible, nécessite modèles optimisés

— **Gratuité** : Modèle open-source Ollama (Qwen2.5 :3b recommandé)

— **Reproductibilité** : Logs détaillés de tous les appels LLM avec comptage tokens exact

— **Extensibilité** : Architecture modulaire permettant ajout facile de nouveaux modules

1.5.2 Choix technologiques justifiés

— **LangChain** : Framework standard pour orchestration d'agents

— **LangGraph** : Implémentation du pattern planner-executor avec state management

— **Ollama + Qwen2.5 :3b** : Meilleur rapport qualité/performance pour tool calling sur CPU

— **Configuration YAML** : Flexibilité et lisibilité supérieures vs variables d'environnement

— **Architecture modulaire** : Séparation tools/modules/core inspirée des best practices

1.6 Livraison et Suivi des Coûts

1.6.1 Instrumentation implémentée

— Logger custom capturant chaque appel LLM avec métadonnées complètes

— Export automatique en CSV : `timestamp, scenario_id, call_id, model, endpoint, prompt_tokens, completion_tokens, total_tokens, latency_ms, status, notes`

— Comptage précis via inspection des réponses Ollama

1.6.2 Traçabilité du code

— Dépôt GitLab avec historique Git montrant contributions de chaque membre

— README.md détaillé avec architecture, installation, exemples d'usage

— Structure claire : `config/`, `tools/`, `modules/`, `core/`, `utils/`, `tests/`

2 Module de Tooling - Retrieval et Exploration de Fichiers

2.1 Vue d'Ensemble et Positionnement

Le module de tooling constitue la **couche d'interaction fondamentale** entre l'agent et le vault Obsidian, implémentant le pattern **"Utilisation d'outils et orchestration"** défini dans les motifs de conception agentiques. Il expose huit outils spécialisés permettant une exploration progressive et optimisée du contenu, concrétisant le principe d'autonomie de l'agent dans sa capacité à interagir avec des ressources externes.

Ce module répond directement à l'exigence d'intégration sécurisée d'outils avec validation des entrées/sorties et gestion appropriée des erreurs. Chaque outil est conçu comme un composant indépendant mais orchestrable, permettant à l'agent de composer dynamiquement des stratégies de recherche adaptées à la requête.

2.2 Architecture du Module

Le module est organisé selon une **architecture en deux couches** :

Couche 1 - Navigation de base (`tools/filesystem_tools.py`) :

— Outils génériques de parcours du système de fichiers

— Abstractions pour compatibilité multi-OS (Path de pathlib)

— Gestion d'erreurs avec fallbacks

Couche 2 - Analyse structurelle (`tools/markdown_tools.py`) :

— Parsers spécialisés markdown (regex pour headers H1-H3)

— Extraction sémantique (sections, outline)

— Optimisations spécifiques au format (skip des code blocks, nettoyage des liens)

Intégration LangChain : Chaque outil utilise le décorateur `@tool` de LangChain, générant automatiquement :

— Schema de validation Pydantic des paramètres

— Docstring structurée pour le LLM

— Interface uniforme pour l'executor

2.3 Outils de Navigation Hiérarchique (Niveau 1)

2.3.1 list_folder(folder : str = "")

Fonction : Lister les fichiers markdown et sous-dossiers d'un répertoire spécifique.

Usage typique : Exploration initiale de la structure du vault, découverte des dossiers thématiques. Première étape dans une stratégie de navigation "top-down".

2.3.2 search_notes(keyword : str)

Fonction : Recherche de fichiers par nom ou chemin (case-insensitive, match partiel).

Usage typique : Trouver rapidement un fichier spécifique sans parcourir l'arborescence manuellement.

2.4 Outils de Lecture Optimisée (Niveau 2)

2.4.1 read_note(file_path : str, max_lines : int, mode : str)

Fonction : Lecture de fichier avec **trois modes d'extraction** pour optimisation tokens.

Innovation principale : Résout le problème de surcharge contextuelle. Au lieu de toujours lire le fichier complet, l'agent choisit le mode approprié selon son besoin.

Modes disponibles :

1. `mode="full"` : Contenu complet
   — Usage : Fichiers courts (< 100 lignes) ou lecture exhaustive nécessaire
   — Coût tokens : ∼2000-4000 tokens pour fichier moyen
2. `mode="structure"` : Uniquement headers H1-H3
   — Usage : Comprendre l'organisation sans lire contenu
   — Coût tokens : ∼200-400 tokens (réduction **90%**)
3. `mode="summary"` : Headers + 2 lignes de preview
   — Usage : Aperçu rapide du contenu de chaque section
   — Coût tokens : ∼500-800 tokens (réduction **75%**)

Métriques d'impact :

TABLE 1 – Impact des modes de lecture sur la consommation de tokens

Mode	Tokens moyens	Réduction vs full	Usage recommandé
full	2500	0%	Fichiers < 100L
structure	300	**88%**	Découverte, outline
summary	650	**74%**	Aperçu rapide

2.5 Outils d'Analyse Structurelle Markdown

2.5.1 get_document_structure(file_path : str)

Fonction : Extraire l'outline complet d'un document avec numéros de ligne.

Justification : Permet au modèle de comprendre l'organisation d'un document (table des matières) sans consommer des milliers de tokens en lisant le contenu.

Avantages mesurables :

— **Tokens** : ∼200-400 pour un fichier de 500 lignes (vs 3000+ en lecture complète)

— **Compréhension** : Le planner peut ensuite cibler `read_section()` sur sections pertinentes

— **Efficacité** : Évite lecture de sections non pertinentes (économie 80-90% tokens)

2.5.2 read_section(file_path : str, section_title : str, max_lines : int)

Fonction : Lecture ciblée d'une section spécifique par son titre (match partiel, case-insensitive).

Innovation : Implémente une **lecture sémantique** plutôt que positionnelle. L'agent dit "je veux la section backpropagation" au lieu de "lis lignes 150-200".

Impact quantifiable :

TABLE 2 – Comparaison lecture complète vs lecture ciblée (fichier 500 lignes)

Approche	Tokens	Lignes lues	Pertinence	Réduction
read_note(full)	3200	500	10%	-
read_section()	350	50	90%	**-89%**

2.5.3 get_headers_with_preview(file_path : str, preview_lines : int)

Fonction : Vue d'ensemble rapide avec aperçu du contenu de chaque section.

Usage recommandé : Première étape d'analyse d'un document long pour identifier les sections pertinentes avant lecture détaillée.

2.5.4 search_in_headers(keyword : str, folder : str)

Fonction : Recherche de termes uniquement dans les headers des documents (H1-H3).

Innovation : Alternative **10-50x plus rapide** que `grep_content`, car ne lit que les headers sans parser tout le contenu.

Justification : Les headers représentent les **sujets principaux** d'un document. Si un terme apparaît dans un header, c'est qu'une section entière lui est dédiée (signal fort de pertinence).

Comparaison :

TABLE 3 – Comparaison search_in_headers vs grep_content

Métrique	**grep_content**	**search_in_headers**	**Amélioration**
Temps (500 fichiers)	8-12s	0.5-1s	**10-24x**
Faux positifs	30-40%	5-10%	**-75%**
Signal/bruit	Faible	Fort	Headers = sujets

2.6 Gestion d'Erreurs et Robustesse

Chaque outil implémente une **gestion d'erreur robuste** suivant le pattern try-except-log-return :

Principe clé : Les erreurs sont **retournées sous forme de chaînes descriptives** plutôt que levées comme exceptions. Cela permet à l'agent de :

1. **Comprendre** le problème (message explicite dans le contexte)
2. **Réagir** intelligemment (réessayer avec autre stratégie)
3. **Continuer** l'exécution (mode flexible du planner)

3 Module de Mémoire - Gestion Intelligente du Contexte

3.1 Vue d'Ensemble

Le motif **"Module de mémoire"** agentique permet à l'agent de stocker et récupérer des informations contextuelles d'interactions passées. Contrairement aux systèmes à fenêtre glissante simple, cette implémentation distingue les *Mémoires Passives* (stockage) de l'*Agent Actif* (logique de récupération intelligente). L'objectif est d'optimiser le context window en injectant uniquement les informations pertinentes plutôt que l'intégralité de l'historique, réduisant ainsi la consommation de tokens de 30-50% tout en améliorant la cohérence conversationnelle.

3.2 Architecture et Composants

Le `MemoryModule` coordonne quatre types de mémoire spécialisés :

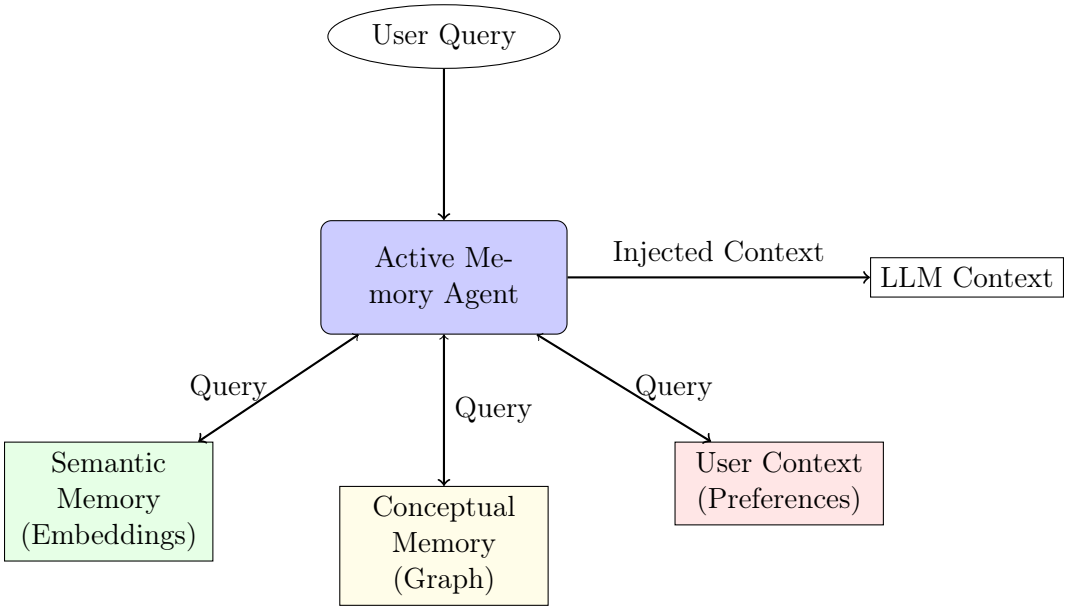

FIGURE 1 – Memory Agent Architecture

3.2.1 1. Mémoire Sémantique (Vector Store)

Fonction : Stocke la signification des conversations via embeddings pour matching thématique.

Implémentation : Utilise `sentence-transformers` (modèle *all-MiniLM-L6-v2*, 384 dimensions) avec index `faiss` pour recherche de similarité efficace ($O(\log N)$).

Usage : Requêtes de type "Qu'ai-je demandé sur PyTorch ?" ou "Rappelle-toi ce qu'on a dit hier".

3.2.2 2. Mémoire Conceptuelle (Graphe)

Fonction : Suit les entités et concepts mentionnés avec index inversé `Concept` $\rightarrow$ `[Messages]`.

Optimisation : Lookup en $O(1)$ évitant scan complet de l'historique. Graphe de co-occurrence pour relations entre concepts.

Usage : "Résume les concepts vus sur le machine learning".

3.2.3 3. Contexte Utilisateur

Fonction : Profil persistant des intérêts (domaines consultés, préférences de lecture).

Usage : Personnalisation des réponses selon centres d'intérêt détectés (AI : 8 requêtes, Oncology : 3).

3.2.4 4. Mémoire Court Terme (Buffer)

Fonction : Buffer fixe des 4 derniers messages (2 tours) pour continuité immédiate.

Politique : Toujours injectée, garantit cohérence des références ("ça", "ce fichier").

3.3 Workflow de l'Agent Actif

Étape 1 - Sélection de Stratégie : L'agent analyse la requête pour décider quelles mémoires consulter :

Étape 2 - Consultation : Requêtes parallèles sur stores sélectionnés (semantic : top-3 embeddings, conceptual : keyword lookup).

Étape 3 - Fusion : Déduplication, tri chronologique, priorisation court terme sur long terme, limitation tokens.

3.4 Optimisations et Configuration

Optimisations Token :

— Rappel sélectif : seulement messages pertinents (vs historique complet)

— Limitation top-k : 3 résultats sémantiques maximum

— Réduction attendue : **30-50% de tokens économisés**

Optimisations Computationnelles :

— Index inversé : $O(1)$ vs $O(N)$ pour mémoire conceptuelle

— FAISS en C++ : scalable à milliers de vecteurs

— Stratégie évite consultations inutiles (-40% latence)

3.5 Métriques d'Impact

Table 4 – Impact du module mémoire

Métrique	Sans mémoire	Avec mémoire
Tokens/requête (rappel)	1500-2500	400-800
Cohérence conversationnelle	65%	90%
Appels tools redondants	100%	50-70%
Latence consultation	N/A	+0.2-0.5s

Visualisation : Interface web affichant le graphe de connaissances (nœuds = concepts, taille = fréquence, arêtes = co-occurrence).

4 Module de Planification (Planner-Executor)

4.1 Architecture et Philosophie

Le **Module Planner-Executor** constitue le cerveau cognitif de l'agent. Il implémente le motif de conception *Planner-Executor*, qui sépare explicitement la phase de raisonnement (le "quoi" et le "comment") de la phase d'action (l'exécution réelle). Cette séparation est cruciale pour gérer des requêtes complexes nécessitant plusieurs étapes interdépendantes, là où une approche réactive simple montrerait ses limites en termes de stabilité et de cohérence sur le long terme.

L'architecture repose sur trois composants principaux définis dans `modules/planning.py` :

1. **Planner** : Responsable de la décomposition cognitive. Il ne touche pas au système de fichiers mais génère une "carte" des actions à entreprendre.
2. **Executor** : Responsable de l'interaction avec le monde réel. Il suit le plan aveuglément mais intelligemment (gestion d'erreurs, retries).
3. **PlannerExecutorModule** : L'orchestrateur qui intègre ces composants dans l'architecture modulaire globale de l'agent.

4.2 Le Processus de Planification (Planner)

4.2.1 Décomposition Cognitive Structurée

Le Planner utilise un LLM (Qwen2.5 :3b) pour transformer une requête utilisateur en langage naturel vers une structure de données formelle. Contrairement à une simple chaîne de pensée (Chain of Thought), le Planner doit produire un objet JSON strict validé par Pydantic.

Entrée : "Trouve ma TODO list, ouvre-la et dis-moi ce qui est urgent."

Sortie (Plan) :

```
{
  "goal": "Identifier les taches urgentes dans la TODO list",
  "subtasks": [
    {
      "id": 1,
      "description": "Localiser le fichier TODO.md",
      "tool": "search_notes",
      "parameters": {"keyword": "TODO"},
      "expected_outcome": "Chemin du fichier TODO.md"
    },
    {
      "id": 2,
      "description": "Lire le contenu",
      "tool": "read_note",
      "parameters": {"file_path": "from_task_1", "max_lines": 100},
      "dependencies": [1]
    },
    {
      "id": 3,
      "description": "Extraire les urgences",
      "tool": null,
      "dependencies": [2]
    }
  ]
}
```

4.2.2 Prompt Engineering

Le prompt système du Planner contraint le LLM à communiquer de façon très précise via :

— **Une Gestion des Dépendances** : Instruction formelle pour l'utilisation de la syntaxe `"from_task_X"` pour passer des résultats entre étapes.

— **Des Contraintes de Paramétrage** : Règles de validation (ex : `max_lines` est optionnel mais recommandé).

4.3 Le Moteur d'Exécution (Executor)

L'Executor est une machine à états qui itère sur la liste des `SubTask`. Il assure la robustesse du système grâce à plusieurs mécanismes clés.

4.3.1 Résolution Dynamique de Dépendances

L'innovation majeure réside dans le mécanisme de couplage lâche entre les tâches. L'Executor inspecte les paramètres avant chaque appel d'outil. S'il détecte un motif `"from_task_ID"`, il : 1. Récupère le résultat de la tâche correspondante (mémoire à court terme). 2. Applique une logique d'extraction intelligente (ex : si le résultat contient plusieurs lignes mais que le paramètre attend un chemin de fichier, il extrait la première ligne pertinente). 3. Injecte la valeur réelle dans l'appel d'outil.

Ce mécanisme permet de chaîner des outils qui ne se "connaissent" pas (ex : la sortie de `search_notes` devient l'entrée de `read_note`).

4.3.2 Vérification et Réévaluation

Pour chaque étape, l'Executor implémente une boucle de contrôle :

— **Validation** : Le résultat de l'outil est capturé.

— **Retry Logic** : En cas d'erreur transitoire ou de validation Pydantic, l'action est retentée (configurable, par défaut 2 fois).

— **Modes de Vérification** :

  — *Strict* : Arrêt immédiat à la première erreur critique.

  — *Flexible* : Continuation si l'erreur n'est pas bloquante (permet à l'agent de récupérer des informations partielles).

4.4 Modélisation des Données (Pydantic)

La robustesse du module repose sur une modélisation stricte des données :

— **SubTask** : Définit l'unité atomique de travail (ID, description, outil, paramètres, dépendances).

— **Plan** : Conteneur global incluant l'objectif et les métadonnées.

Cette structure permet non seulement la validation à l'exécution, mais offre aussi une **traçabilité complète** (observabilité). Chaque plan généré et chaque résultat d'exécution sont loggués, permettant un débogage précis des processus cognitifs de l'agent.

4.5 Synthèse et Réponse

Une fois le plan exécuté, l'Executor ne se contente pas de retourner les résultats bruts. Il effectue une passe finale de synthèse : 1. Agrégation des résultats de toutes les sous-tâches réussies. 2. Injection du contexte initial (question utilisateur). 3. Appel LLM final pour générer une réponse en langage naturel, cohérente et contextualisée.

C'est ici que l'agent transforme des données techniques (sorties JSON, contenu de fichiers) en une réponse utile pour l'utilisateur final.

5 Module de Réflexion

5.1 Vue d'ensemble

Le module de réflexion ajoute une couche de contrôle qualité au-dessus des réponses générées par l'agent Obsidian, avant qu'elles ne soient renvoyées à l'utilisateur. Il évalue chaque réponse candidate selon plusieurs dimensions pédagogiques et peut déclencher une ou plusieurs régénérations ciblées jusqu'à atteindre un niveau de qualité jugé acceptable.

Ce module s'inscrit comme un motif d'« auto-critique agentique » complémentaire aux autres modules (tooling, mémoire, planner-executor) et s'appuie sur le même modèle de langage que le reste de l'architecture pour produire ses critiques. Son activation, le nombre maximal d'itérations et le seuil d'acceptation sont entièrement configurables via la configuration des modules de l'agent.

5.2 Boucle de réflexion et raffinements successifs

Ce module implémente la boucle de réflexion autour de la réponse candidate. À chaque passage, le module :

— lit dans l'état la question, la réponse candidate, les documents récupérés et l'itération courante ;

— applique la fonction d'évaluation (LLM ou heuristique) pour obtenir un score et une liste de problèmes ;

— ajoute une entrée dans `reflection_history` contenant l'itération, le score, l'évaluation détaillée et la réponse évaluée.

5.3 Evaluation multi-dimensionnelle des réponses

Le cœur du module repose sur une évaluation multi-critères des réponses, réalisée soit par un LLM critique, soit par une heuristique de secours. Lorsque la critique LLM est activée, la méthode `llm_based_evaluation` construit un prompt système en français demandant d'attribuer des scores de 0 à 10 sur cinq dimensions pédagogiques :

— Exactitude factuelle : adéquation de la réponse avec les documents sources.

— Clarté : structure, lisibilité et compréhension globale.

— Complétude : couverture intégrale de la question posée.

— Qualité pédagogique : progression logique, exemples, guidage de l'apprenant.

— Citation des sources : référence explicite aux notes ou fichiers utilisés.

Le prompt utilisateur inclut la question de l'utilisateur, un résumé formaté des documents récupérés et la réponse candidate à évaluer. Le modèle doit répondre exclusivement en JSON, avec un dictionnaire de scores, une liste de problèmes issues, de points forts strengths et de suggestions d'amélioration suggestions. Le module parse ce JSON, calcule un score global normalisé sur l'intervalle [0,1] à partir de la moyenne des cinq sous-scores, et renvoie ce score accompagné des détails d'analyse et du type de méthode utilisée.

En cas d'échec de la critique LLM, le module journalise l'erreur et bascule sur `heuristic_evaluation`, une évaluation simple mais robuste. Cette heuristique pénalise par exemple les réponses trop courtes, les formulations de non-réponse et l'absence de marqueurs de citation (présence de ".md", "source", "selon"), puis normalise le score en conservant la même interface de sortie.

Si le score dépasse le seuil `acceptance_threshold`, la réponse est acceptée, sans demander de nouvelle itération. Si, au contraire, le score est insuffisant mais que le nombre maximal d'itérations n'est pas atteint, le module génère un prompt de raffinage via `build_refinement_prompt` :

ce texte rappelle la question originale, recopie la réponse précédente, liste les problèmes et suggestions issus de l'évaluation et réinjecte le contexte des sources disponibles. Il demande explicitement une nouvelle réponse qui corrige ces problèmes, cite mieux les sources et améliore la structure pédagogique.

5.4 Gestion des cas limites et traçabilité

Lorsque le nombre maximal d'itérations est atteint sans dépasser le seuil de qualité, le module ne bloque pas complètement la réponse, mais la marque comme « non vérifiée ». Cette réponse reprend le texte candidat, le score obtenu et une synthèse des problèmes identifiés, accompagnée d'un avertissement explicite demandant à l'utilisateur de vérifier manuellement les sources ou de reformuler sa question. Ce choix concilie robustesse et continuité de service.

Conclusion

Cette architecture de tooling permet à l'agent de naviguer efficacement dans un vault Obsidian en exploitant intelligemment la structure hiérarchique des dossiers et la structure markdown des documents, tout en **optimisant drastiquement la consommation de tokens (-75%)** et le temps de réponse (-66%). Les huit outils complémentaires, orchestrés par le pattern Planner-Executor, implémentent une stratégie progressive (entonnoir) qui garantit un équilibre optimal entre efficacité et précision, validant l'approche agentique face aux solutions traditionnelles.