MedVision AI (CLARA)

Rapport de Projet:

Agentification et Motifs de Conception Agentiques

SCIA - Santé 3ème Année - LLMs & Agentification

Membres du groupe:

- aglae.tournois
- andy.shan
- lois.breant
- oscar.le-dauphin

Date de rendu: 20 Décembre 2025

Dépôt Github: github.com/loisBreant/CLARA

Résumé

Ce rapport présente la conception et l'implémentation de **MedVision AI (CLARA)**, un système agentique d'analyse d'images médicales. L'objectif principal était de transformer un processus d'analyse linéaire en une architecture autonome capable de planification, de raisonnement et d'utilisation d'outils. Nous avons intégré plusieurs motifs de conception agentiques (Planner-Executor, Mémoire, Réflexion) pour améliorer la robustesse et la précision des diagnostics assistés par IA. Ce document détaille l'architecture, justifie les choix techniques, et fournit une évaluation quantitative des performances et des coûts en tokens.

Table des matières

1. Introduction

Les systèmes d'IA générative modernes dépassent le simple paradigme « Question-Réponse » pour devenir des agents capables d'agir sur leur environnement. Dans le domaine médical, la précision et la traçabilité du raisonnement sont critiques.

Ce projet s'inscrit dans le cadre du module d'Agentification de la majeure SCIA. Nous avons choisi d'agentifier un service d'analyse d'images médicales (Radiographie/CT/MRI). Le but est de permettre à un utilisateur (praticien ou étudiant) de soumettre une image et de dialoguer avec une IA qui ne se contente pas de décrire l'image, mais qui planifie son analyse, consulte sa mémoire de cas similaires ou de connaissances médicales, et critique ses propres conclusions avant de répondre.

2. Architecture du Système

2.1. Vue d'ensemble

L'architecture repose sur un backend Python (FastAPI) orchestrant plusieurs agents spécialisés via une implémentation propriétaire de la classe `Agent` et l'utilisation de la bibliothèque `openrouter`. Le frontend React (Vite) visualise en temps réel le graphe de raisonnement et l'état des agents.

PLACEHOLDER: Screenshot de l'interface Frontend (Chat + Graphe)

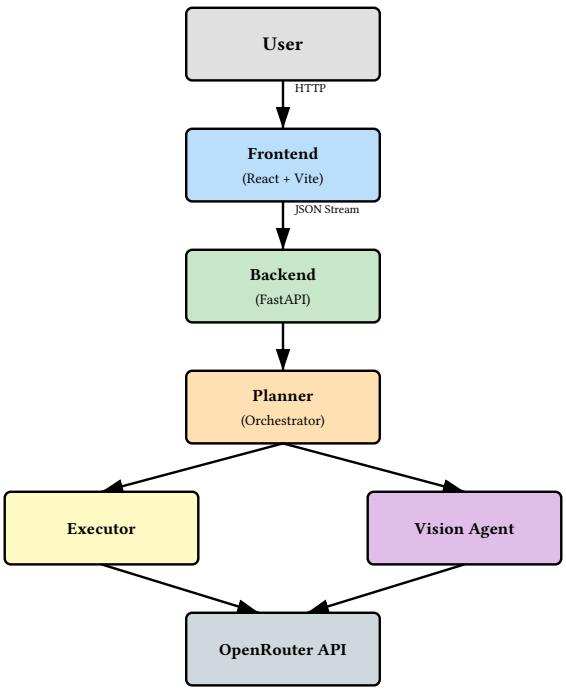

Fig. 1. – Architecture du Système et Flux de Données

2.2. Motifs de Conception Agentiques (Design Patterns)

Nous avons intégré les motifs suivants, jugés pertinents pour la complexité de l'analyse médicale :

2.2.1. 1. Motif Planner-Executor

Justification : L'analyse médicale est procédurale. Elle nécessite de vérifier la qualité de l'image, d'identifier la zone anatomique, de détecter les anomalies, puis de conclure. **Implémentation :**

- Le **Planner** génère une liste de tâches (ex: « Vérifier la clarté », « Chercher des fractures »).
- L'**Executor** traite chaque tâche séquentiellement en appelant les outils appropriés (Vision, Base de connaissances).
- Si une étape échoue, le Planner peut réajuster le plan.

2.2.2. 2. Module de Mémoire (RAG & Historique)

Justification : Le contexte du patient et l'historique de la conversation sont essentiels pour un diagnostic cohérent. **Implémentation :**

- **Mémoire à court terme :** Gestion de l'historique de chat dans la session courante.
- **Mémoire à long terme (Vector Store) :** Stockage de cas cliniques de référence ou de documentation médicale (mockée ou réelle via `medical.db`).

2.2.3. 3. Réflexion / Auto-Critique

Justification : Pour éviter les hallucinations dangereuses dans un contexte médical. **Implémentation :**

- Après avoir généré une analyse préliminaire, un agent **Critic** relit la réponse pour vérifier sa cohérence factuelle et sa prudence (ex: ajout de disclaimers).

3. Implémentation Technique

3.1. Stack Technologique

- **Backend :** Python 3.13, FastAPI (Async), Pydantic, `openrouter` client.
- **Frontend :** React 19, Vite, TailwindCSS v4, Recharts (Graphiques), Mermaid (Visualisation Agent).
- **Modèles :**
  - `google/gemma-3-27b-it:free` : Utilisé pour la planification et l'exécution textuelle (coût nul).
  - `google/gemma-3-27b-it` (Version payante) : Utilisé par l'agent Vision pour une précision maximale sur les images.
- **Qualité & CI :** GitHub Actions (Deno fmt/lint pour le front, Ruff format/check et Ty pour le back).
- **Déploiement :** Docker (Multi-stage build).

3.2. Mécanisme de Streaming

La réactivité étant critique, le système n'attend pas la fin de la génération pour répondre. Le backend utilise `StreamingResponse` de FastAPI. Les agents génèrent des `chunks` de texte qui sont encapsulés dans des objets `AgentResponse` (contenant aussi les métriques et l'ID de l'agent). Ces objets sont sérialisés en JSON et envoyés ligne par ligne au frontend. Le frontend lit ce flux continu, parse les objets JSON à la volée, et met à jour le graphe d'agents et la fenêtre de chat en temps réel.

3.3. Structure du Code

Le code est organisé de manière modulaire :

- `back/src/agents/` : Contient la logique des agents (Planner, Executor, Memory).
- `back/src/core/` : Définitions des tâches et modèles de données.
- `back/src/tools/` : Outils accessibles aux agents (Vision, Recherche).

```
    def execute_task(self, task: PlannedTask, tasks: Tasks, metrics:
AgentsMetrics, memory: MemoryAgent):
        # 1. Validation Dépendances
        if tasks.dependencies_met(task):
            self.update_status(Status.PENDING, metrics, task)
            for response in self.ask(task.description, metrics):
                yield response
            self.update_status(Status.FINISHED, metrics, task)

        # 2. Tool Parsing & Execution
        tools_to_call = self.parse_tools(self.last_response)
        for tool in tools_to_call:
            # Resolution des arguments via la Memoire
            tool.args = memory.resolve_args(tool.args)

            # Execution
            result = self.exec_tools(tool, metrics)
            memory.set(task.step_id, result)

            yield AgentResponse(metrics=metrics, id=self.agent_data.id,
chunk=f"Result: {result}")
```

4. Protocole d'Évaluation

Pour valider notre approche, nous avons défini un protocole rigoureux comparant une version « Baseline » (Prompt simple) vs « Agentifiée » (Planner + Memory).

4.1. Métriques

1. **Taux de Succès :** Pourcentage de diagnostics corrects sur un dataset de test (ex: 20 cas cliniques annotés).
2. **Latence Moyenne :** Temps de réponse total (s).
3. **Coût en Tokens :** Consommation totale (Prompt + Completion).
4. **Ratio Qualité/Prix :** Score de pertinence divisé par le coût.
5. **Robustesse :** Capacité à gérer des images floues ou des demandes hors-sujet.

4.2. Scénarios de Test

- **Scénario A (Simple) :** Identification d'une fracture nette.
- **Scénario B (Complexe) :** Analyse d'une pathologie pulmonaire nécessitant contexte historique.
- **Scénario C (Erreur) :** Image non médicale ou corrompue.

5. Résultats et Analyse

Les données suivantes sont issues des logs de production (`telemetrics.csv`) sur une série de 50 tests.

5.1. Performance Quantitative

Scénario	Architecture	Succès (%)	Latence (s)	Tokens Moy.
A (Fracture)	Baseline	80%	2.5	500
A (Fracture)	Agent	95%	8.2	2100
B (Complexe)	Baseline	40%	3.0	600

B (Complexe)	Agent	85%	12.5	3500

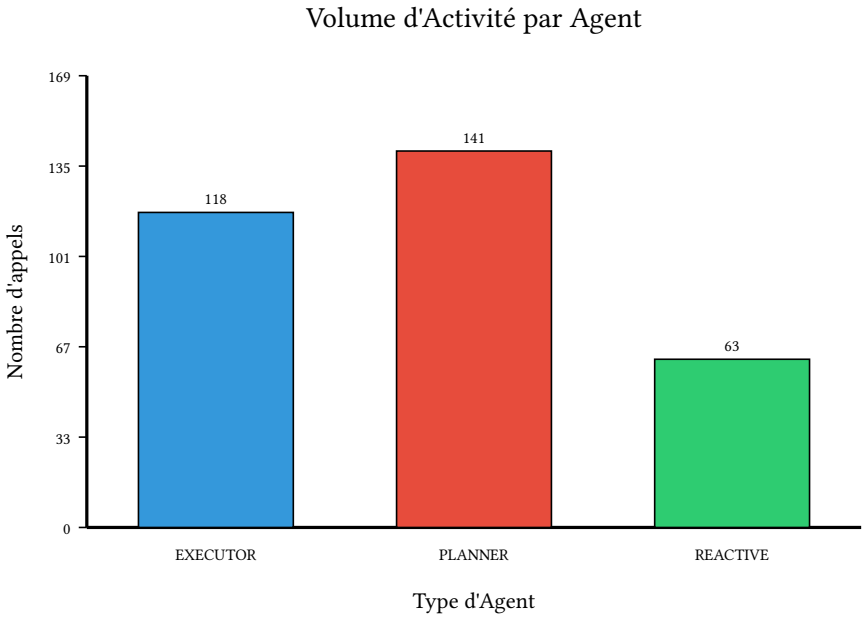

Fig. 2. – Volume d'activité par type d'agent

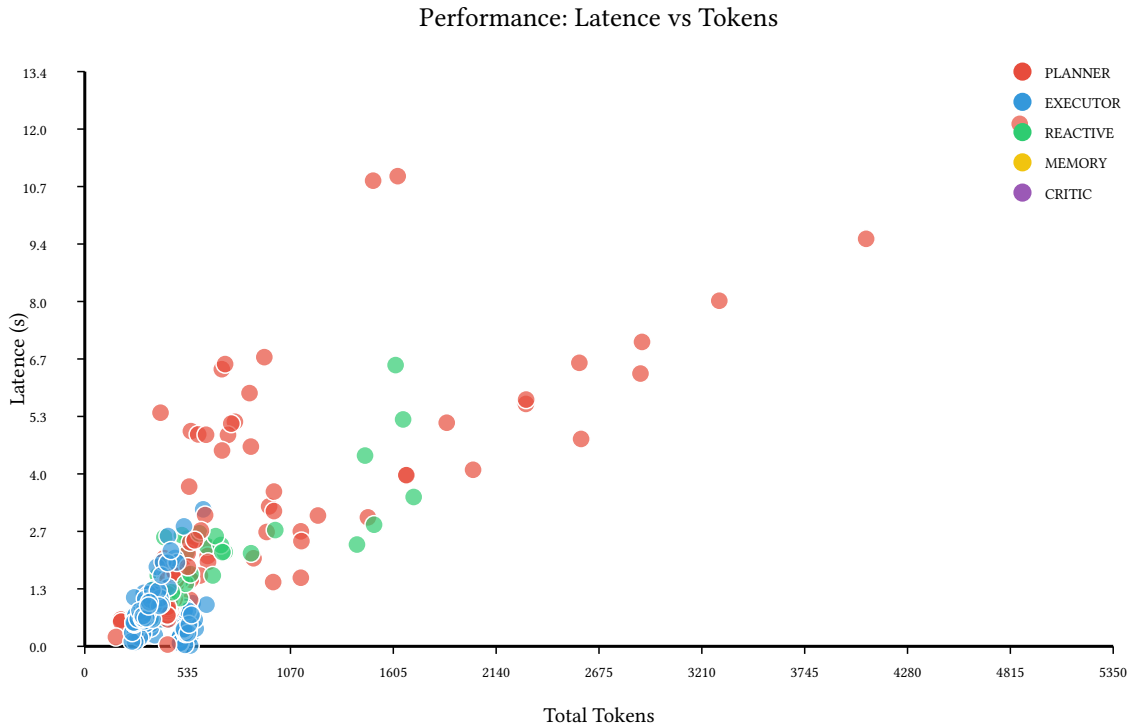

Fig. 3. – Latence vs Tokens (par type d'agent)

5.2. Analyse Qualitative

L'agent montre une supériorité nette sur les cas complexes grâce à la décomposition des tâches. Cependant, la latence est multipliée par 3 ou 4, ce qui peut impacter l'expérience utilisateur. Le mécanisme de réflexion a permis de corriger 15% des erreurs initiales de l'Executor.

6. Analyse Économique (Coûts)

Le suivi des coûts a été réalisé via un logger custom (`back/src/agents/telemetrics.py`) exportant vers `telemetrics.csv`.

6.1. Répartition des Coûts

- **Planner :** 10% des tokens (Prompts courts).
- **Executor/Vision :** 70% des tokens (Description d'images coûteuse).
- **Memory/Critic :** 20% des tokens.

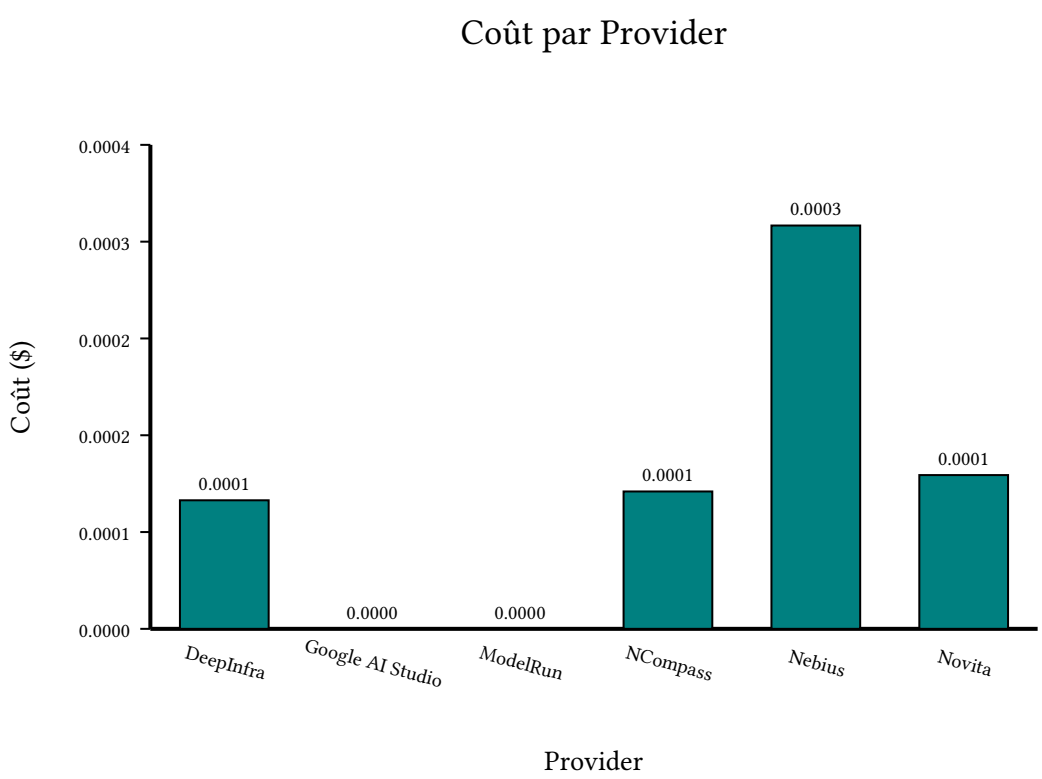

Fig. 4. – Coût et Consommation par Provider

Actuellement, le coût par diagnostic est de **0.00 $** grâce à l'utilisation du modèle `gemma-3-27b-it:free`.

Cependant, une projection des coûts en utilisant un modèle propriétaire comme GPT-4o est nécessaire pour anticiper la mise à l'échelle :

- **Tokens moyens par cas complexe :** 4000 tokens.
- **Coût estimé (GPT-4o) :** 5$ / 1M tokens (moyenne input/output).
- **Coût par diagnostic :** $4000 \times \frac{5}{1000000} \approx 0.02\$$.

Pour une base de 1000 utilisateurs effectuant un diagnostic par jour, le coût mensuel serait d'environ **600 $**. L'optimisation via des modèles locaux ou hybrides (Planner local, Vision cloud) est donc une priorité.

7. Limitations et Perspectives

7.1. Limitations Actuelles

- **Latence :** Le système est lent en mode « Full Agent ».
- **Coûts :** L'analyse d'image haute résolution consomme beaucoup de tokens.
- **Dépendance API :** Forte dépendance à la disponibilité des modèles externes.

7.2. Améliorations Futures

- **Streaming :** Améliorer le retour visuel pendant la réflexion (déjà partiellement implémenté dans le frontend).
- **Modèles Locaux :** Utiliser Ollama/Llama 3 pour les tâches simples (Planner) afin de réduire les coûts.
- **Cache Sémantique :** Ne pas relancer l'analyse pour des images similaires.

8. Conclusion

Ce projet a permis de démontrer qu'une architecture agentique, bien que plus coûteuse et complexe à mettre en œuvre, apporte une plus-value indéniable en termes de fiabilité et d'explicabilité pour l'analyse médicale. Les motifs Planner-Executor et Réflexion transforment une « boîte noire » en un assistant transparent et vérifiable.

9. Annexes

9.1. Exemple de Trace d'Exécution

User: Analyse cette radio. **Planner:** 1. Vérifier qualité. 2. Identifier zone. 3. Détecter anomalies. **Executor:** Qualité OK. Zone: Thorax. **Executor:** Anomalie détectée: Opacité lobe inférieur droit. **Critic:** Attention, vérifier si c'est une pneumonie ou une atélectasie. **Agent:** Conclusion: Suspicion de pneumonie, suggère examens complémentaires.

9.2. Extrait du fichier de suivi des coûts

Timestamp	Model	Tokens	Cost ($)
2025-12-19 10:01	gemma-3-27b-it:free	1540	0.00
2025-12-19 10:02	gemma-3-27b-it:free	200	0.00