

CLARA (Clinical Logic & Agentic Reasoning Assistant)

Rapport de Projet:

Agentification et Motifs de Conception Agentiques

ING3 SCIA & Santé - LLM'S AGENTIC AND BIOMEDICAL

Membres du groupe:

- aglae.tournois (Santé)
- andy.shan (SCIA)
- lois.breant (SCIA)
- oscar.le-dauphin (SCIA)

Date de rendu: 20 Décembre 2025

Dépôt Github: github.com/loisBreant/CLARA

Résumé

Ce rapport présente la conception et l'implémentation de **CLARA (Clinical Logic & Agentic Reasoning Assistant)**, un système agentique d'analyse d'images médicales. L'objectif principal est 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

Résumé	1
1. Introduction	3
2. Architecture du Système	3
2.1. Vue d'ensemble	3
2.2. Design Patterns de Conception Agentiques	4
2.2.1. Motif Planner-Executor	4
2.2.2. Module de Mémoire (RAG & Historique)	4
2.2.3. Module de Réflexion / Auto-Critique	4
3. Implémentation Technique	4
3.1. Stack Technologique	4
3.2. Mécanisme de Streaming	4
3.3. Structure du Code	4
4. Analyse Économique	5
4.1. Répartition des Coûts	5
5. Limitations et Perspectives	6
5.1. Limitations Actuelles	6
5.2. Améliorations Futures	6
6. Conclusion	7
7. Annexes	8
7.1. Exemple de Trace d'Exécution	8
7.2. Extrait du fichier de suivi des coûts	8

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 des majeures SCIA et Santé. 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)

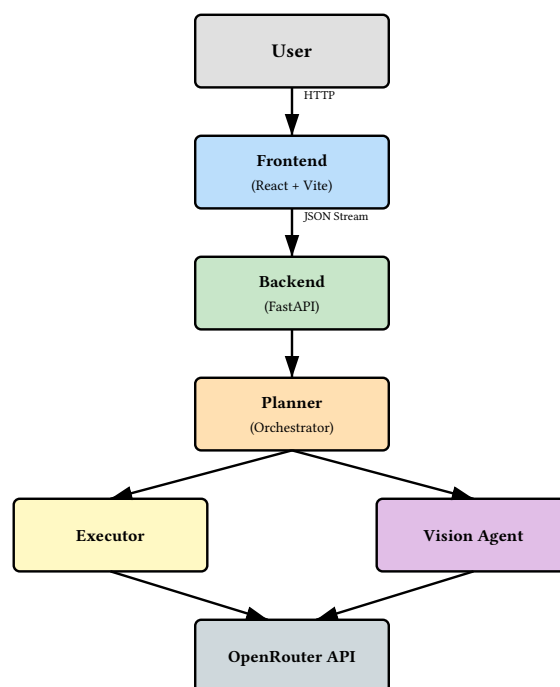


Fig. 1. – Architecture du Système et Flux de Données

2.2. Design Patterns de Conception Agentiques

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

2.2.1. 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 (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. 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 : Stockage de cas cliniques de référence ou de documentation médicale mockée ou réelle via `medical.db`.

2.2.3. Module de Réflexion / Auto-Critique

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

Implémentation :

- Après avoir généré une analyse préliminaire, un agent 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 (aucun coût).
 - `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):
    # 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)

    # Tool parsing & execution
    tools_to_call = self.parse_tools(self.last_response)
    for tool in tools_to_call:
        # Résolution des arguments via la mémoire
        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. Analyse Économique

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

4.1. Répartition des Coûts

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

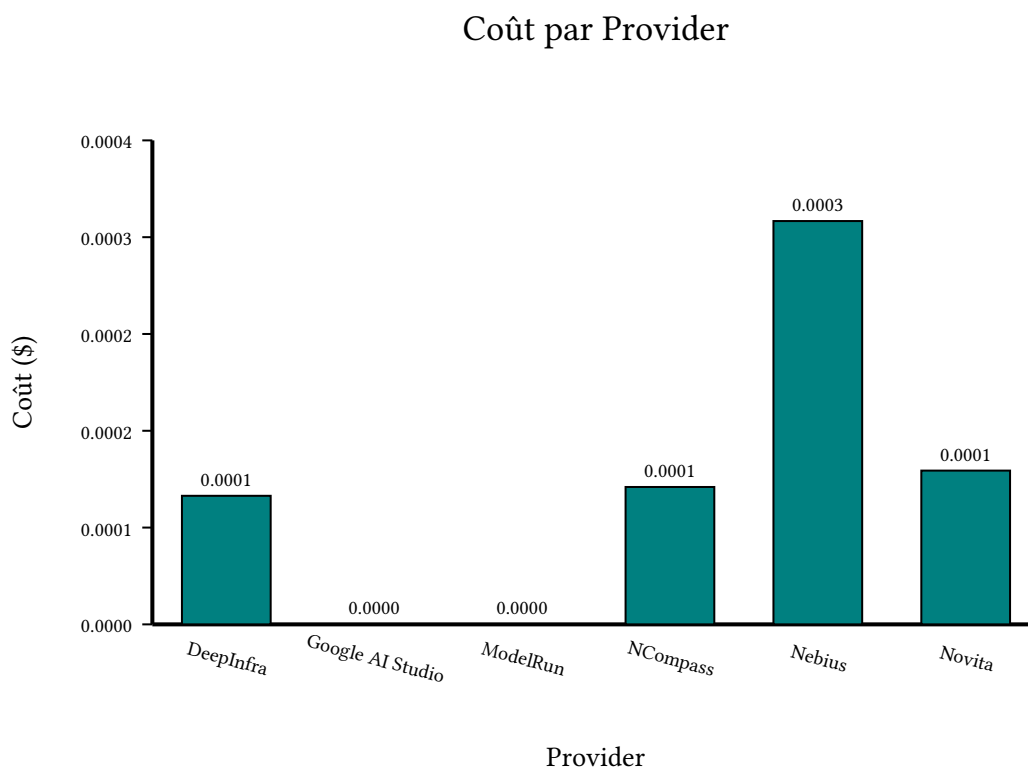


Fig. 2. – Coût et Consommation par Provider

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 payant 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.

5. Limitations et Perspectives

5.1. 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.

5.2. Améliorations Futures

- **Streaming** : Améliorer le retour visuel pendant la réflexion (déjà partiellement implémenté dans le frontend).
- **Modèles Locaux** : Utiliser Ollama pour les tâches simples (Planner) afin de réduire les coûts.
- **Optimisation des Prompts** : Affiner les prompts pour réduire la consommation de tokens.
- **Augmentation de la Mémoire** : Intégrer une base de connaissances médicale plus riche.
- **Cache Sémantique** : Ne pas relancer l'analyse pour des images similaires.

6. 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.

7. Annexes

7.1. 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.

Agent: Conclusion: Suspicion de pneumonie, suggère examens complémentaires.

7.2. 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