
✅ **Objectif :**

Développer une **application IA local-first** avec des agents spécialisés (ex : SEO, copywriting, image generation…) s'appuyant sur des **modèles exécutés nativement sur ton Mac M4**, sans cloud, tout en gardant une architecture modulaire et évolutive. On ajoutera plus tard la possibilité d'utiliser des modèles dans le cloud (chatGPT, Mistral, …), mais l'architecture doit être pensée dès le début à offrir les 2 possibilités : Local et Cloud.

🧱 **2.**

Architecture logicielle idéale

Composant	**Choix recommandé**
Langage principal	Python 3.11+ (meilleure compatibilité, async natif, très riche pour IA)
Front-end	Next.js, Vue.js ou React (si besoin d'interface graphique moderne)
Back-end orchestration	FastAPI (léger, async, parfait pour API REST locales ou orchestrer des agents)
Serveur LLM local	LM Studio ou Ollama (selon format) + API OpenAI-compatible
Modèles LLM	Mistral 7B Instruct, LLaMA 3 8B, Phi-3, en format .gguf ou via Ollama
Agent framework	CrewAI, AutoGen, ou architecture maison en Python avec asyncio
Vector DB local	ChromaDB ou Weaviate (en local pour RAG / recherche sémantique)
Base de données	SQLite pour la simplicité ou PostgreSQL pour scalabilité
Interface CLI/Dev	Langchain ou LLamaIndex (pour structurer prompts, pipelines, et agents)
UI pour testing rapide	Chatbot UI connecté à LM Studio en localhost

---

🧠 **3.**

Flux de développement recommandé

1. **Choisis un orchestrateur d'agents** (CrewAI)
2. **Configure Ollama** pour charger ton modèle GGUF localement
3. **Teste via API locale** compatible OpenAI (http://localhost:1234/v1)
4. **Développe une couche API en FastAPI** qui sert de chef d'orchestre entre les appels LLM, les bases de données, les logs, et les réponses
5. **Ajoute un front-end (React/Vue)** si besoin de GUI
6. **Ajoute un moteur RAG local (Chroma)** si tu veux contextualiser avec documents
7. **Optionnel : ajoute une couche "UI admin"** pour observer logs, prompts, réponses des agents

---

🧩 **5.**

Exemples d'outils à combiner localement

- **LangChain** : si tu veux structurer prompts, outils, mémoire contextuelle
- **Ollama** : pour charger modèles très facilement et appeler /v1 API
- **ChromaDB** : base vectorielle 100% locale et légère
- **CrewAI** : gestion d'agents avec rôles, outils et coordination
- **LLamaIndex** : pipeline RAG complet et compatible avec Chroma/Ollama

---

🧪 **En bonus : environnement de test local**

- Utilise **Chatbot UI** ou **Open WebUI** connecté à ton API locale pour interagir manuellement avec tes LLMs
- Utilise **Weights & Biases** ou **Langfuse** pour tracer les performances si besoin

---

Voici l'**organigramme des interactions entre modules** pour une architecture multi-agent locale utilisant **FastAPI** et **Ollama**.

🧠 Explication des rôles :

- 👤 **Utilisateur** : interagit via l'interface web
- 💻 **Frontend** : envoie les requêtes à l'API (FastAPI)
- 🚀 **API Orchestrateur (FastAPI)** : route les requêtes vers le gestionnaire d'agents
- 🧠 **Gestionnaire d'agents** : décide quels agents activer en fonction du besoin
- ✍️ **Agent Rédaction** : génère du contenu (blogs, posts…) via LLM
- 📈 **Agent SEO** : analyse et optimise le contenu avec LLM
- 📚 **Agent RAG** : récupère de l'information contextuelle (via Chroma) puis l'enrichit avec le LLM
- 🤖 **Ollama** : exécute les modèles LLM en local (Mistral, LLaMA…)
- 🗃️ **ChromaDB** : base vectorielle pour la recherche sémantique locale

Ajouter un vrai moteur RAG avec **ChromaDB** ou **LlamaIndex** ?

Créer un **front-end** minimal en React pour interagir avec l'API ?

Ajouter un **log des interactions** (journal des prompts/réponses) ? Voici ce qui a été ajouté à ton projet :

✅ **ChromaDB** intégré pour gérer un moteur vectoriel local (moteur RAG).

✅ **Agent RAG** capable de retrouver du contexte depuis Chroma pour enrichir ses réponses.

✅ **Logging local** simple dans un fichier logs.txt qui trace chaque interaction (prompt + réponse).

📂 **Script d'ingestion**

ingest_docs.py

Permet d'extraire automatiquement le texte de fichiers PDF déposés dans un dossier ./docs

Utilise unstructured et pdfminer.six pour parser

Les extraits sont ajoutés à **ChromaDB** pour que l'agent RAG puisse les exploiter

---

✅ Le front-end React intègre:

1. 🧭 **Sélecteur d'agent manuel** (Auto, SEO, Rédaction, RAG)
2. 🗂️ **Historique des interactions** (prompt, agent utilisé, réponse)

Chaque message envoyé est tracé avec son agent dans une liste consultable sous le champ de réponse.

ajouter :

- la **possibilité d'effacer l'historique**
- une **sauvegarde locale de l'historique** via localStorage

Parfait. Le composant ChatInterface.jsx est maintenant le cœur orchestrateur de l'interface.

Voici la prochaine étape :

1. PromptInput.jsx → champ de saisie du prompt
2. AgentSelector.jsx → menu déroulant de sélection d'agent
3. ResponseBox.jsx → zone d'affichage de la réponse
4. HistoryPanel.jsx → affichage de l'historique
5. utils/api.js → envoi des requêtes à l'API
6. utils/history.js → gestion du localStorage