

Chapitre 8 : Les Agents IA

Le grand fantasme de l'informatique est de pouvoir automatiser des tâches que les humains n'ont aucune envie de réaliser. La démocratisation des LLMs et leur capacité de suivre des instructions en few-shot a permis d'automatiser un nombre de tâches de plus en plus important (résumé, extraction ciblée d'information, rédaction de documents administratifs...) sans avoir besoin de créer un système informatique spécifique. Néanmoins comme vu au chapitre précédent, les LLMs ne sont rien d'autre que des modèles d'IA capables de prédire le mot suivant et ils ont leur limitations, notamment leur manque d'accès à des sources de données spécifiques, ce qui limite drastiquement les tâches réelles que l'on peut automatiser avec. De nombreux travaux ont donc été menés depuis pour augmenter les capacités de raisonnement des modèles¹ ou leur donner la capacité d'utiliser des outils extérieurs (tool use). De nombreux travaux, dès 2023 ont réfléchi à la façon d'intégrer des LLM dans des systèmes capables de traiter des tâches plus complexes (résolution d'issues github, QA challengeant) de façon autonome, ce qui implique plutôt un travail d'ingénierie logicielle encadrant les appels de LLMs plutôt que du travail sur les modèles eux-mêmes (bien que souvent on les fine-tune pour que cela fonctionne mieux). On a donné à ces systèmes le nom d'« agent » et cela a donné lieu en 2023 à un ensemble de travaux hétéroclites avant qu'une vraie définition et des guides de bonnes pratiques de commencent à émerger. 2024 a été une année de stabilisation de la définition et de l'architecture des agents, et tout laisse à penser que 2025 sera l'année de leur début de mise en production / de leur industrialisation. Mais que sont vraiment ces agents IA dont les influenceurs parlent tant ? Comment fait-on pour en construire et quels sont les composants à prendre en compte ? Quels tâches sont-ils effectivement capables d'automatiser et quelles sont leurs limites ? C'est ce que l'on tentera de définir dans les pages suivantes !

I) Les Agents IA : une définition contemporaine

Avec humour, on peut voir les agents IA comme étant une étape intermédiaire entre des fonctionnaires (terme RH d'« agent ») et l'AGI (Artificial General Intelligence, un système d'IA qui aurait des capacités cognitives équivalentes ou supérieures à celles d'un être humain sur tous les plans). L'objectif d'un agent informatique est de réaliser une tâche précise avec la même dextérité que le ferait un agent humain pour pouvoir le décharger de cette mission. Si la mission est à peu près claire, la définition technique ne l'est pas du tout et a pris plus d'un an à être élaborée.

a) Les agents informatiques : background historique

La communauté scientifique a fini par désigner par le terme d'« agent IA » des systèmes qui font appel à des LLMs pour automatiser des tâches complexes. Cette automatisation va au-delà de la création d'un prompt spécifique et nécessite souvent des traitements en amont et en aval de(s)

1 <https://arxiv.org/pdf/2502.21321>

appel(s) à des LLM. Ce n'est finalement que réactiver un terme déjà introduit dans les années 90 sans que l'on puisse attribuer le terme à un auteur en particulier.

Histoire sémantique de l'agent :

Le terme agent désigne simplement un système qui fait quelque chose.

Le terme est un dérivé du participe présent du verbe agir en latin (agens, agentis) et signifie « celui qui fait, agit, produit, met en œuvre ». Pour être plus précis, le terme est issu de la scolastique médiévale (physique / métaphysique) et désigne une force ou un élément capable de modifier un autre élément, qui exerce une action par opposition à ce qui la subit (cf aussi agent au sens grammatical vs le patient). Le mot a pris le sens plus tard (vocabulaire officiel et politique) d'une personne chargée d'une mission pour devenir synonyme de chargé de mission ou de fonctionnaire. En informatique, le mot a pris le sens précis de logiciel envoyé sur un réseau pour effectuer une tâche à la place de l'utilisateur et sans son intervention. Cette notion a pris de l'importance avec le web sémantique quand on a mis en place des programmes chargés d'effectuer des recherches à la place d'humains (crawlers, bots...). Un agent dit « mobile » peut se déplacer d'un site à l'autre et accéder à des données ou des ressources. Il se déplace tout en conservant son code, mais aussi son état d'exécution et décide de façon autonome son action suivante. Parmi ces types d'agents logiciel on peut citer l'agent système d'un système d'exploitation, qui tourne en tâche de fond, les robots d'indexation, l'agent utilisateur qui gère l'interface entre le navigateur et le web... Un agent en informatique désigne donc un programme dont l'architecture est telle qu'il gère tout seul son exécution sans supervision. Toutefois le terme se rapproche d'autres notions importantes en informatique.

Les automates :

Nos programmes informatiques sont des scripts chargés d'exécuter une tâche et ils sont peu adaptables. Le plus souvent, plusieurs cas de figure (dépendant des interactions du programme avec l'utilisateur ou l'environnement) sont prévus, notamment avec des systèmes de routages ou qui énumèrent les actions possibles. Finalement on peut représenter un programme complexe comme un graphe où chaque nœud représente un état d'exécution du programme, et chaque arête l'exécution de certaines actions qui vont modifier cet état. Certaines des bibliothèques implémentant des agents IA, comme on le verra plus tard, se sont basées sur des intuitions similaires. Mais cette manière de concevoir et de représenter un système informatique censé automatiser une tâche est en fait héritée d'un des concepts d'informatique fondamentale, les automates.

Un automate est un dispositif reproduisant en autonomie une séquence d'actions prédéterminées sans intervention humaine²

Les automates finis, ou plus généralement les automates d'états, qui sont des modèles mathématiques utilisés pour décrire le comportement de systèmes discrets. Dans ces modèles, un système peut se trouver dans un nombre fini d'états, et passer de l'un à l'autre en réponse à des événements ou des entrées. Un programme peut ainsi être vu comme un automate déterministe, où chaque état représente une configuration spécifique du programme (par exemple, une étape d'un formulaire, un écran dans une application, ou une condition logique particulière), et chaque

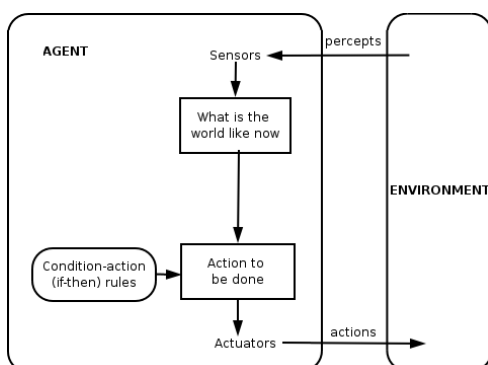
² Définition de wikipédia. Cf étymologie du mot <https://bailly.app/automatos> c'est ironique car un automate signifie à l'origine « qui pense par lui-même » alors que l'automate est prédéterminé, sans notion de pensée ou de raisonnement.

transition correspond à une action, une entrée utilisateur ou une condition logique remplie. C'est ce paradigme qui sous-tend notamment le développement de machines à états dans les systèmes embarqués, les jeux vidéo, ou encore certaines architectures logicielles.

Mais cette conception, bien que puissante, montre rapidement ses limites dès que le système devient complexe ou doit faire face à des environnements dynamiques et partiellement inconnus. Dans ce cas, il devient difficile, voire impossible, de prédéfinir toutes les transitions possibles ou d'anticiper tous les scénarios. C'est précisément là qu'interviennent les agents intelligents. Un agent IA ne se contente pas d'exécuter un script figé : il observe, évalue, décide, et agit dans un environnement parfois incertain. Autrement dit, si l'on comparait un programme classique à un automate, un agent IA serait une entité capable de modifier dynamiquement sa propre structure d'automate, d'ajouter ou retirer des états, ou même de changer sa stratégie de transition en cours d'exécution. Ce changement de paradigme – d'un système figé à un système adaptatif – est ce qui distingue fondamentalement les agents intelligents des scripts traditionnels, mais il n'en est que l'évolution naturelle. On ajoute la capacité de raisonnement / d'auto planification.

Les programmes / systèmes multi-agents :

Une autre notion d'informatique à laquelle les agents IA font échos est celle des systèmes multi-agents. Un système multi-agent est un système composé d'un ensemble d'agent (au sens large, peut désigner un programme comme un utilisateur) actifs dans un certain environnement avec lequel ils interagissent selon des règles données. Chaque agent est (du moins partiellement) autonome et il n'y a pas pilotage du système global (par opposition aux systèmes de



type « hive mind » dans lesquels les agents sont coordonnés). Chaque agent interagit avec l'environnement³ et décide d'effectuer son action. Par exemple dans un système multi-agent chargé de gérer le bon fonctionnement d'une serre ou d'un vivarium, le thermostat est un agent qui peut (via un capteur thermomètre) mesurer la température de la serre (observation de l'environnement), et selon des règles ou des instructions décider d'activer les systèmes de régulation de température (action sur l'environnement)

Souvent les systèmes multi agents ont été utilisés pour modéliser l'évolution naturelle de sociétés humaines ou de populations animales, avec des applications parfois aux sciences humaines comme la sociologie et surtout l'économie ou sont utilisés dans certaines industries comme le jeu vidéo (gestion des Personnages Non Joueurs, NPC) .

Concevoir un système multi agent aujourd'hui implique plusieurs rôles essentiels de l'IA :

- *nécessité de l'IA pour gérer la prise de décision de l'agent.
- * nécessité d'IA (voire IA distribuée) pour la distribution de l'exécution des tâches entre des agents.

En d'autres termes, la planification et l'orchestration de la tâche par les agents.

Cela a trait tout au tant avec le génie logiciel pour pouvoir penser une architecture efficace et bien distribuée capable de faire tourner tous ces agents.

La gestion d'agents pose en effet des questions techniques importantes qu'il convient de résoudre

3 https://en.wikipedia.org/wiki/Intelligent_agent

- * comment des agents peuvent-ils agir en même temps sur le même environnement partagé ? (question d'accès aux ressources partagées, de choix dynamique, de planification)
- * Comment gérer l'ordre de priorité d'exécution des agents entre eux ?
- * Comment adapter dynamiquement leur programmation aux évolutions de l'environnement ? (les informations de l'environnement changent la réaction des agents selon leur protocole, mais devrait peut être aussi modifier le protocole des agents eux-mêmes.)

Ainsi l'implémentation des agents IA remet sur le devant de la scène des problématiques d'ingénierie logicielle de ce domaine « assez niche » qui avait perdu en popularité.

La définition des agents IA telle qu'elle s'est cristallisée aujourd'hui est directement l'héritière de différentes branches de l'informatique, que les « AI practitioners » en aient conscience ou non, et cela influence grandement l'implémentation des agents qui en est faite, surtout quand il est question de gestion des workflows et d'architectures multi-agents.

b) Les différents degrés de l'agentivité : une définition graduelle

Malgré ces rappels historiques nécessaires, on n'est pas encore parvenu à une définition satisfaisante de ce qu'est un agent IA. Cette définition n'a pas été facile à construire car les travaux des chercheurs ont exploré des directions parfois très différentes. On est toutefois parvenu à une ébauche de consensus. Si on compare quelques définitions :

- * "An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through actuators. (Stuart Russell and Peter Norvig, dans un livre fondateur qui a été écrit avant l'ère de l'IA générative)
- * "an Agent is: an AI model capable of reasoning, planning, and interacting with its environment." (Hugging Face)
- * "An artificial intelligence (AI) agent refers to a system or program that is capable of autonomously performing tasks on behalf of a user or another system by designing its workflow and utilizing available tools." (IBM)
- "A generative AI system is considered an agent when it goes beyond creating output based on a prompt and begins to act on information to achieve a desired outcome." (Babak Hodjat, CTO of AI at Cognizant)

Toutes ces définitions, bien qu'elles décrivent de façon correcte un agent IA son tout de même assez floues. L'agent est essentiellement défini par ses capacités (définitions constructiviste). Il peut lire son environnement, raisonner sur son observation, le modifier. Implémenter des agents signifie donc implémenter de nouvelles capacités⁴. On le définit aussi comme allant au-delà des capacités génératives du LLM (définition par la négative).

Arriver à une définition simple et unique ne semble pas possible. À partir de quel degré d'autonomie parle t'on d'agent ? Donner des informations contextuelles dans un prompt n'est-il pas déjà faire interagir son LLM avec son environnement ? Puisque les capacités des LLMs évoluent, comment la frontière entre LLM simple et agent évolue-t-elle ? Mon message est qu'il n'y a rien d'extrêmement stable ni de satisfaisant pour l'instant.

4 <https://arxiv.org/pdf/2308.11432>

Au lieu de voir une frontière stricte entre LLM simple et agent, il est bien plus intéressant de considérer que les systèmes informatique dans lesquels on intègre des appels de LLM présentent différents degrés d'agentivité⁵.

Agency Level	Description	How that's called	Example Pattern
☆☆☆	LLM output has no impact on program flow	Simple Processor	<code>process_llm_output(llm_response)</code>
★☆☆	LLM output determines an if/else switch	Router	<code>if llm_decision(): path_a() else: path_b()</code>
★★☆	LLM output determines function execution	Tool Caller	<code>run_function(llm_chosen_tool, llm_chosen_args)</code>
★★★	LLM output controls iteration and program continuation	Multi-step Agent	<code>while llm_should_continue(): execute_next_step()</code>
★★★	One agentic workflow can start another agentic workflow	Multi-Agent	<code>if llm_trigger(): execute_agent()</code>

* On part du LLM simple qui n'est capable que d'une simple transformation de l'entrée qu'il reçoit en une sortie textuelle.

* On peut utiliser le LLM comme un routeur dynamique : au lieu de reposer dans un système sur des branchages explicites (if/else) pour décider dans quelle branche aller ensuite, on peut déléguer au LLM le choix et donc créer des systèmes plus complexes, s'adaptant par exemple à des données hétérogènes (par exemple un LLM peut analyser le contenu d'un document et voir si certaines conditions sont satisfaites (présence d'une date de naissance, moins de 26 ans...) pour pouvoir choisir l'étape suivante. Le LLM n'a pas a proprement parler d'interaction avec son environnement mais est capable de décisions dynamiques.

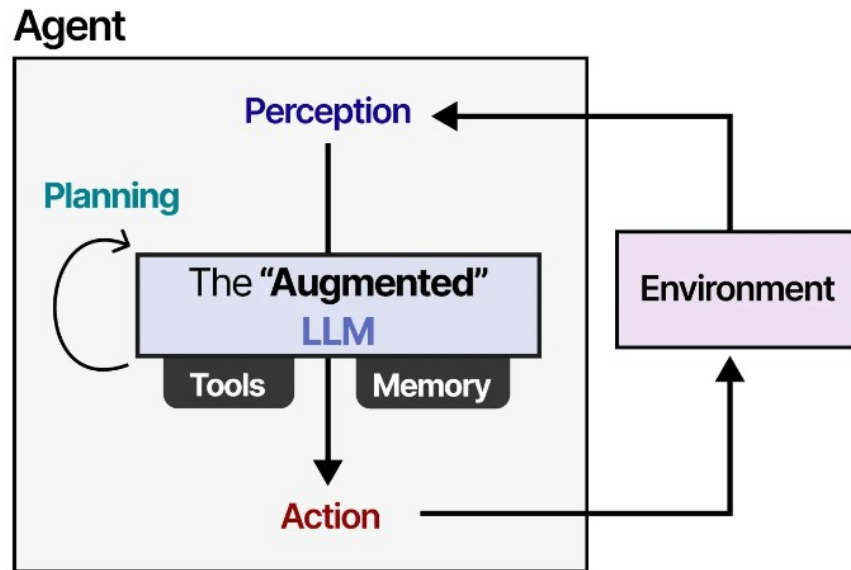
* Le LLM peut décider selon l'état actuel du traitement de l'étape suivante qu'il faut suivre (voir II.1). L'agent commence à avoir une certaine autonomie, mais il est limité aux actions qu'on l'autorise à effectuer.

* Le LLM peut à partir de la tâche générale décider des action à effectuer ensuite. Ces actions, via des mécanismes à implémenter, seront ensuite effectuées. Puis à partir des résultats collectés le LLM reprend la main et ce jusqu'à ce qu'un état de terminaison satisfaisant soit atteint. On exploite donc pleinement les capacités de raisonnement et de planification des LLMs dans l'exécution de la tâche.

* Il est possible de créer des systèmes ou plusieurs agent qui ont des rôles particuliers communiquent et s'envoient des informations pour réaliser la tâche. Par exemple pour automatiser le screening de CV, un agent pourrait être chargé de lire les CV et d'extraire les compétences, un autre agent de vérifier la correspondance entre des compétences d'un candidat et une fiche de poste, et un agent « manager » d'orchestrer tout cela... Des fonctions d'orchestration s'ajoutent à l'utilisation des LLM pour planifier une marche à suivre.

l'on peut faire entre un workflow, ou un LLM, parfois doté d'outils est employé, mais ou l'exécution des étapes est monotone, et un agent. On parle d'agent conversationnel lorsque l'input

5 https://huggingface.co/docs/smolagents/en/conceptual_guides/intro_agents



Si on reprend la définition de départ de Russel et Norvig, on peut schématiser un agent ainsi, avec en son centre un LLM augmenté (cf II) de capacité pour percevoir et agir sur son environnement.

En absence de définition ferme, on peut voir l'agentivité comme un spectre, jugé selon l'autonomie du système à prédire les prochaines actions qu'il va effectuer. C'est la principale distinction que d'un utilisateur est prise en considération en entrée. Un agent conversationnel sera un agent à proprement parler si et seulement si le système choisit dynamiquement quelle suite d'actions effectuer après chaque tour de parole.

Les frontières sont poreuses puisqu'il s'agit au final de construire des systèmes à partir de blocs existants (fonctions, LLMs...). Les workflows / agents sont des structures composables (un agent peut devenir un composant d'un autre agent). Dès lors le recyclage est important. Il y a deux choses qui important et peuvent faire évoluer un système dans le spectre de l'agentivité : les composants et l'architecture.

c) Autant de définitions que d'implémentations

Si les agents doivent tous réaliser la même chose (gérer l'interface entre des LLMs et un environnement pour automatiser des tâches complexes, les stratégies mises en place pour y parvenir son différentes : comment stocker les états intermédiaires du traitement ? Comment penser la communication d'un composant à l'autre ? Comment organiser les composants de l'agent ? ... Des traitements différents de ces problématiques techniques ont donné naissance à différentes implémentations de systèmes agentiques qui ont des caractéristiques différente.

C'est un écosystème qui reste encore en mutation et de nouvelles librairies sont proposées régulièrement. Ces implémentations concurrentes se complètent et se répondent. Si certains favorisent la simplicité (smolagents), d'autres la capacité à traiter des tâches très abstraites (crewAI). Enfin quand on s'aventure dans le monde des agents, il faut partir d'acquis construits par la communauté et ne pas chercher à réinventer la roue. Beaucoup de code technique a été écrit (pour gérer les sorties fautives de LLM, le maximum d'essais possibles...) et le refaire demanderait des efforts significatifs. Il n'est pas bon pour autant de se rendre esclave entièrement d'un framework et de ses évolutions.

Tableau comparatif des frameworks d'agents LLM

Framework	Caractéristiques clés	Avantages	Inconvénients
LangChain / LangGraph	<ul style="list-style-type: none"> - Python, modulaire, orienté chaîne - LangGraph : orchestrations en graphe - Outils, mémoire, planning - Compatible multi-LLM 	<ul style="list-style-type: none"> + Hyper flexible + Énorme écosystème (LangSmith, outils) + Multi-agent possible via LangGraph + Très populaire 	<ul style="list-style-type: none"> - Complexe à prendre en main - Verbose - Risque d'overhead pour de petits projets
Semantic Kernel	<ul style="list-style-type: none"> - C# et Python - Architecture par plugins (skills) - Planner intégré - Fortement orienté productivité/app métier 	<ul style="list-style-type: none"> + Idéal pour intégration dans apps .NET + Support Microsoft + Extensible en entreprise 	<ul style="list-style-type: none"> - Moins riche que LangChain pour la RAG ou agents évolués - Courbe d'apprentissage pour le planner
CrewAI	<ul style="list-style-type: none"> - Python + Studio no-code - Rôles d'agents + tâches - Orchestration séquentielle - +700 intégrations API/outils 	<ul style="list-style-type: none"> + Très productif pour MVP ou entreprise + UI/monitoring intégrés + Bon équilibre code/no-code 	<ul style="list-style-type: none"> - Moins flexible pour orchestration libre - Lourd pour des projets simples - Télémétrie activée par défaut
smol-agents (HF)	<ul style="list-style-type: none"> - Python - Super léger (1 script = 1 agent) - Appelle des fonctions Python avec du code généré - Intégration HF Transformers 	<ul style="list-style-type: none"> + Simplicité extrême + Très customisable + Peu de dépendances + Parfait pour agents locaux ou expérimentaux 	<ul style="list-style-type: none"> - Pas de mémoire native ni de coordination multi-agent - Moins mature - Nécessite de gérer exécution et sécurité soi-même
OpenAI Swarm	<ul style="list-style-type: none"> - Python - Agents + handoffs - Exécution sans état - Pensé client-side / edge 	<ul style="list-style-type: none"> + Minimaliste et pédagogique + Très customisable + Sécurité et vie privée (pas de persistance) 	<ul style="list-style-type: none"> - Pas adapté à la prod - Pas de mémoire longue - Très peu de supervision ou gestion d'erreurs
Autogen (Microsoft)	<ul style="list-style-type: none"> - Python - Communication inter-agents asynchrone - Rôles configurables - Support multi-agent natif + human-in-the-loop 	<ul style="list-style-type: none"> + Très puissant pour coordination + Open-source soutenu par Microsoft + Scalable avec asyncio 	<ul style="list-style-type: none"> - Complexité élevée - Prise en main difficile - Peu d'UI / monitoring hors intégration perso



On présentera brièvement quelques unes des librairies les plus utilisées ainsi que leurs caractéristiques spécifiques. De cette bref comparatif, que je vous laisse bien sur compléter par des recherches personnelles, on peut déduire plusieurs points importants :

- * Après une période de flottements et d'implémentations autonomes, un appel a été fait pour revenir à la simplicité et à un code qui pouvait être lu intégralement avant d'être utilisé tel quel⁶. Ce constat d'Anthropic a été corroboré par les travaux d'Hugging Face.

- * Tous ces frameworks n'ont pas les mêmes objectifs, il faut veiller à comparer ce qui est comparable

- * Des bouleversements de ce panorama restent possible avec l'arrivée de nouveautés comme les agents dotés de vision ou le Computer Use...

- * On constate un tradeof logique entre simplicité / interprétabilité du framework et complexité des tâches qu'il est capable de prendre en charge. Les frameworks comme Langchain ou CrewAI promettent d'automatiser des tâches vraiment complexes, mais sont assez opaques, coûteux en appels de LLM consommés et l'utilisateur perd complètement la main sur ce qui se passe pendant le traitement. Au contraire les librairies comme smol agents offrent moins de capacités mais plus d'explicailité et de compréhension sur les étapes du processus.

- * Il y a également un tradeof entre la simplicité du code à écrire et le contrôle qu'a l'utilisateur sur ce qu'il fait.

- * Aucune de ces solutions n'est parfaite ni complète. On constate par exemple un manque de réflexion sur la gestion de la mémoire dans la plupart des frameworks, alors que c'est une question importante pour la création de chatbots par exemple.

- * Ces solutions apparaissent comme complémentaires. Langgraph permet une bonne gestion des frameworks en les représentant comme les graphes, crewAI a poussé plus loin la réflexion sur le multi-agent, smolagents est plus interprétable et maléable...

Il faut donc adapter le choix de la librairie au projet pour lequel on veut utiliser des agents.

2) Les composants essentiels : la recette de l'agent réussi

L'implémentation d'agents IA relève plus de l'ingénierie logicielle que de l'intelligence artificielle à proprement parler. Un certain nombre de questions techniques sont à résoudre si on veut que les agents soient capables de réaliser des tâches complexes :

- * Comment donner de la visibilité à l'agent sur son environnement et comment lui permettre d'agir dynamiquement sur lui ? Cette problématique a été gérée par la notion de tool-use

- * Comment un agent peut-il suivre le déroulement d'une mission complexe ? Cela nécessite une notion de persistance et d'effet des actions précédentes.

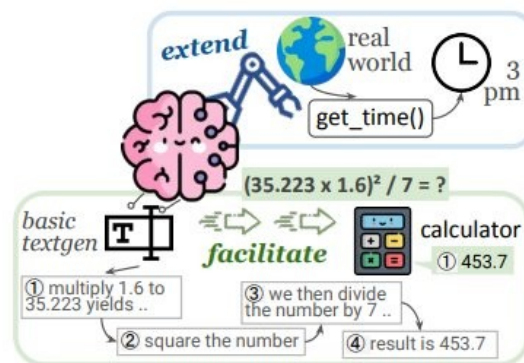
- * Que faire lorsque la complexité de la tâche augmente ? Comment modifier l'implémentation de l'agent pour le rendre capable au mieux de gérer cette difficulté.

6 <https://www.anthropic.com/engineering/building-effective-agents>

a) Lire et agir sur son environnement : la notion de tool use

Les LLMs sont capables de générer du texte et d'effectuer correctement un certain nombre de tâches comme le résumé et pour quelques langues la traduction, mais ils atteignent vite leurs limites sur des sujets tels que le raisonnement mathématique, la réponse à des questions sur des connaissances qui dépassent leurs capacités d'entraînement ...

D'où l'idée de venir compléter ces capacités génératives par d'autres modules. En effet, si un LLM a du mal à correctement répondre à des calculs compliqués, une calculatrice ou un programme python le fait sans problème. L'idée de l'outil consiste donc simplement à faire s'interfacer un LLM capable de générer du texte et d'autres fonctions informatiques qui peuvent faire tout ce qu'il ne sait pas faire correctement. La difficulté est de « faire comprendre » au LLM qu'il a des outils à sa disposition et qu'il puisse apprendre à en faire un bon usage⁷. Les outils utilisés peuvent recouvrir des fonctionnalités très diverses. Les outils les plus utilisés sont évidemment ceux qui permettent aux LLM d'aler requêter des informations, que ce soit sur le web ou dans des bases de données extérieures.








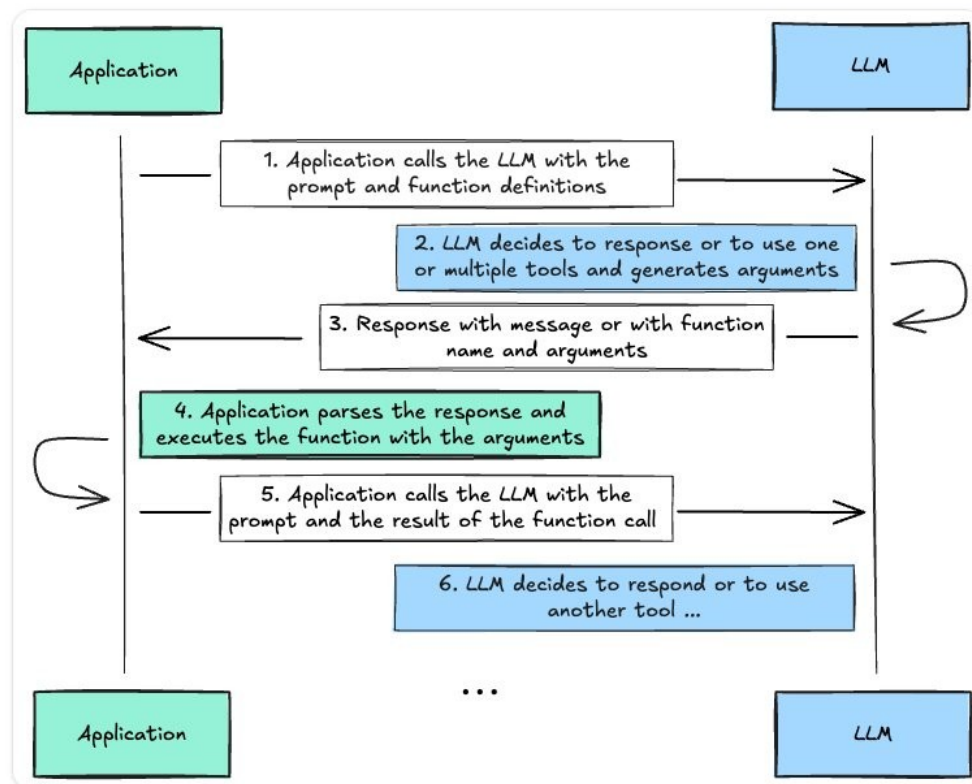
Category	Example Tools
 Knowledge access	<code>sql_executor(query: str) -> answer: any</code> <code>search_engine(query: str) -> document: str</code> <code>retriever(query: str) -> document: str</code>
 Computation activities	<code>calculator(formula: str) -> value: int float</code> <code>python_interpreter(program: str) -> result: any</code> <code>worksheet.insert_row(row: list, index: int) -> None</code>
 Interaction w/ the world	<code>get_weather(city_name: str) -> weather: str</code> <code>get_location(ip: str) -> location: str</code> <code>calendar.fetch_events(date: str) -> events: list</code> <code>email.verify(address: str) -> result: bool</code>
 Non-textual modalities	<code>cat_image.delete(image_id: str) -> None</code> <code>spotify.play_music(name: str) -> None</code> <code>visual_qa(query: str, image: Image) -> answer: str</code>
 Special-skilled LMs	<code>QA(question: str) -> answer: str</code> <code>translation(text: str, language: str) -> text: str</code>

Table 1: Exemplar tools for each category.

Mais comment faire en sorte que le LLM utilise ces outils ? Pour se faire, deux stratégies ont été envisagées. La plus répandue est celle que les créateurs de modèles ont adoptée sous le nom de « function calling » (terme inventé par Open AI⁸)

⁷ <https://arxiv.org/abs/2403.15452>

⁸ <https://platform.openai.com/docs/guides/function-calling?api-mode=chat> désolé je n'arrive pas à retrouver le lien de l'illustration que j'ai utilisée.



L'idée est de laisser comprendre au modèle la liste des outils qu'il a à sa disposition en les lui donnant explicitement dans le system prompt. On donne plus ou moins le docstring de la fonction (description, liste des arguments, de leur type et de celui de la sortie attendue) Puis si le modèle décide si c'est nécessaire d'utiliser un des outils ou s'il doit répondre normalement.

Par exemple si un LLM dispose d'outils nommés `get_weather(location)` et `play_song(song_name)`, si on lui demande « Quel est la météo à Paris », il faut que le modèle choisisse d'exécuter `get_weather(Paris)`. En revanche si on lui demande la date de foncation de l'INALCO il doit pouvoir répondre normalement sans faire appel à un outil.

Pour que le mécanisme de function calling fonctionne, le modèle a subi un fine tuning spécifique l'entraînant à choisir d'exécuter le bon outil dans de nombreux scénarios. Si le modèle décide d'utiliser un outil, il renvoie la plupart du temps un json indiquant le nom de l'outil à utiliser et les arguments à donner, ou avec des balises spécifiques, mais c'est très variable d'un modèle à l'autre. Ainsi lors de l'appel d'un LLM augmenté capable de faire du function calling, plusieurs choses se passent :

- * Si le LLM décide de ne pas utiliser d'outil, il répond dans le format habituel
- * S'il décide d'utiliser un outil, alors (selon le modèle) il répond dans un format structuré donnant le nom de l'outil à utiliser et les éventuels arguments déduits du contexte.
- * Du côté de l'application, le système qui a appelé le LLM doit parser la réponse pour voir si le LLM a choisi un appel d'outil. Si c'est le cas, l'appel doit être exécuté du côté de l'application.

* Le résultat de l'appel d'outil doit être ajouté en tant qu'observation au contexte fourni au LLM. Le LLM doit ensuite être appelé une **seconde fois** et il peut choisir d'appeler un autre outil⁹ ou de fournir une réponse finale à partir de l'observation.

Ce qu'on appelle donc function calling et qui permet d'augmenter les LLM avec des outils n'est rien d'autre qu'un mécanisme transformant plus ou moins le LLM en un routeur disant quand il faut exécuter l'outil. Cette exécution de l'outil est prise en charge par l'application et pas le LLM. Le fait que ce mécanisme soit fiable a été rendu possible par un fine tuning spécifique du modèle. Cela entraîne que :

* Il faut scrupuleusement respecter les conditions de mise en forme (présentation des outils dans le system prompt avec des nouvelles balises spécifiques) sinon cela ne fonctionne pas

* Chaque modèle a utilisé un format différent et il faut donc adapter chaque prompt pour chaque modèle (ce qui est pénible, on utilise autant que possible des utilitaires comme celui de Hugging Face pour faire la conversion de format).

* Il y a aussi des limitations concernant le nombre d'outils que l'on peut utiliser. Des études ont prouvé que à partir d'une dizaine d'outils, le choix perdait en pertinence. Aussi, plus il y a d'outils, plus le system prompt est long et plus la génération va coûter cher. Si on veut faire de son agent un véritable Mister Gadget il faudra soit penser à gérer des agents spécialisés pour des domaines plus précis, avec des boîtes à outils plus précises, soit penser à un mécanisme de sélection d'outils qui en fonction de la requête de l'utilisateur ne donne au LLM que les outils qui risquent de lui être le plus pertinent pour effectuer sa tâche.

Ce mécanisme de function calling implique de mettre en place certaines fonctionnalités du côté de l'application qui utilise le LLM, notamment le parsing de la réponse pour repérer les appels d'outils, leur exécution, le second appel au LLM... Ce n'est toutefois pas la seule possibilité pour pouvoir effectuer du tool use. Il est en effet possible de demander au LLM s'il utilise un outil de générer un code python exécutable. Cette stratégie, retenue par Hugging Face par exemple, a plusieurs avantages¹⁰ :

* Cela permet d'effectuer plusieurs appels d'outil (`get_weather(Paris)` et `get_weather(Marseille)`) en n'effectuant qu'un seul appel de LLM.

* La latence totale du système peut ainsi baisser jusqu'à 30 %

* On évite de plus les erreurs de parsing de json et le respect des noms d'argument.

* Cela augmente le taux de succès (en limitant les risques de propagation d'erreur réalisée lors d'une des étapes intermédiaires).

* L'usage d'outils n'est plus limité aux quelques outils donnés et à leur format attendu mais on peut utiliser avec flexibilité des bibliothèques python pour optimiser la transformation des entrées des outils.

* Le fait d'utiliser du code permet de mieux stocker les étapes intermédiaires (dans des variables) et les passer d'une étape à l'autre du raisonnement.

9 Par exemple si on demande s'il fait meilleur à Paris ou à Marseille, il faudra deux appels de l'outil `get_weather` avant que le LLM ne puisse répondre à partir des observations.

10 <https://arxiv.org/abs/2402.01030>

- * Il y a beaucoup plus de données disponibles pour l'entraînement d'exemples de code pour réaliser des tâches que d'exemples de LLM pour effectuer une génération de json de l'outil à appeler.
- * Le traceback de l'exécution du code permet de corriger d'éventuelles erreurs sans supervision humaine.

Instruction: Determine the most cost-effective country to purchase the smartphone model "CodeAct 1". The countries to consider are the USA, Japan, Germany, and India.

Available APIs

[1] lookup_rates(country: str) -> (float, float)

[2] convert_and_tax(price: float, exchange_rate: float, tax_rate: float) -> float

[3] estimate_final_price(converted_price: float, shipping_cost: float) -> float

[4] lookup_phone_price(model: str, country: str) -> float

[5] estimate_shipping_cost(destination_country: str) -> float

LLM Agent using [Text/JSON] as Action

Think I should calculate the phone price in USD for each country, then find the most cost-effective country.

Action Text: lookup_rates, Germany
JSON: {"tool": "lookup_rates", "country": "Germany"}

Environment 1.1, 0.19

Action Text: lookup_phone_price, CodeAct 1, Germany
JSON: {"tool": "lookup_phone_price", "model": "CodeAct 1", "country": "Germany"}

Environment 780

Action Text: convert_and_tax, 780, 1.1, 0.19
JSON: {"tool": "convert_and_tax", "price": 780, "exchange_rate": 1.1, "tax_rate": 0.19}

Environment 916.3

[... interactions omitted (look up shipping cost and calculate final price) ...]

Action Text: lookup_rates, Japan
JSON: {"tool": "lookup_rates", "country": "Japan"}

[... interactions omitted (calculate final price for all other countries) ...]

Response The most cost-effective country to purchase the smartphone model is Japan with price 984.00 in USD.

CodeAct: LLM Agent using [Code] as Action

Think I should calculate the phone price in USD for each country, then find the most cost-effective country.

Action

```
countries = ['USA', 'Japan', 'Germany', 'India']
final_prices = {}

for country in countries:
    exchange_rate, tax_rate = lookup_rates(country)
    local_price = lookup_phone_price("xAct 1", country)
    converted_price = convert_and_tax(
        local_price, exchange_rate, tax_rate
    )
    shipping_cost = estimate_shipping_cost(country)
    final_price = estimate_final_price(converted_price, shipping_cost)
    final_prices[country] = final_price

most_cost_effective_country = min(final_prices, key=final_prices.get)
most_cost_effective_price = final_prices[most_cost_effective_country]
print(most_cost_effective_country, most_cost_effective_price)
```

Environment 1.1, 0.19

Response The most cost-effective country to purchase the smartphone model is Japan with price 984.00 in USD.

Si cette approche est prometteuse pour avoir des agents IA plus souples et adaptables, il faut prendre en considération les dangers à exécuter un code généré par un LLM et prendre des précautions (en limitant l'accès à certaines librairies, à exécuter le code dans une sandbox sans capacité de nuisance sur le système de fichier...)

Ces mécanismes de tool use, mis en place en adaptant les modèles pour les encourager à utiliser des outils et surtout l'architecture des agents pour les exécuter leur permet d'observer et d'interagir avec leur environnement. Toutefois, il est nécessaire de donner une visibilité au LLM des actions qui ont été préalablement effectuées ce qui implique de gérer intelligemment le contexte donné au LLM pour la génération.

b) La gestion de la mémoire

Une conversation « classique » avec un LLM se présente comme une liste de messages catégorisés comme « system » (le system prompt), « user » (message de l'utilisateur) ou « assistant » (réponse du LLM). Le fait de compléter le system prompt par des description d'outils et de devoir ajouter des observations liées à l'utilisation de ces outils augmente le nombre de tokens consommés. Or la fenêtre de contexte des modèles n'est pas infinie et si l'agent effectue une longue série d'appels d'outil et de raisonnement, au bout d'un moment cette fenêtre de contexte risque d'être saturée.

On retrouve donc pour les agents le même type de problème de mémoire qui sont posés lors de l'implémentation d'agents conversationnels. En effet, si l'utilisateur reste dans la même conversation, au bout d'un moment elle ne rentre plus intégralement dans la fenêtre de contexte. C'est pourquoi même ChatGPT ne donne pas l'intégralité de la conversation comme contexte à la génération suivante, mais en général les derniers messages de la conversation seulement ce qui peut entraîner des oublis problématiques.

Des mécanismes de mémoire plus élaborés ont été conçus pour essayer de limiter ces oublis problématiques en dotant les agents conversationnels de gestionnaires de mémoire (code chargé de gérer au mieux le contenu qui va être donné au LLM dans sa fenêtre de contexte. Parmi les types de mémoire, on peut distinguer :

- * une mémoire à tampon (Conversation Buffer Window Memory) qui ne garde qu'une fenêtre des k derniers messages de la conversation
- * une mémoire par résumé (Conversation Summary Memory). De façon épisodique (tous les 10 ou 20 messages, un LLM est appelé pour résumer tout le début de la conversation, et ce de façon récursive. Cette méthode permet de bien comprimer le contexte, mais entraîne un coût lié à une augmentation du nombre d'appels de LLM et le risque de perdre une information granulaire qui aurait été utile par la suite.
- * Une mémoire reposant sur les graphes de connaissances. A chaque tour de parole, les messages sont analysés et leur contenu vient grossir un graphe. Mais cela implique des coûts computationnels et des questions de fiabilité.
- * Enfin, les messages peuvent être stockés dans un index et être recherchés via une recherche sémantique à partir du dernier message de l'utilisateur pour ne sélectionner que les messages utiles à la génération de la réponse suivante.

Ces divers mécanismes de compression de la conversation sont également transposables à des agents autonomes qui, pour leur bon fonctionnement sont censés avoir à leur disposition une grande quantité d'information de nature hétérogène. Un papier vraiment intéressant (Cognitive Architectures for Language Agents¹¹) essaie de formaliser cette question des informations que doit manipuler un agent IA en s'inspirant des travaux de biologie sur le fonctionnement de la mémoire des animaux. Les auteurs distinguent trois types de mémoire :

Memory Type	What is Stored	Human Example	Agent Example
Semantic	Facts	Things I learned in school	Facts about a user
Episodic	Experiences	Things I did	Past agent actions
Procedural	Instructions	Instincts or motor skills	Agent system prompt

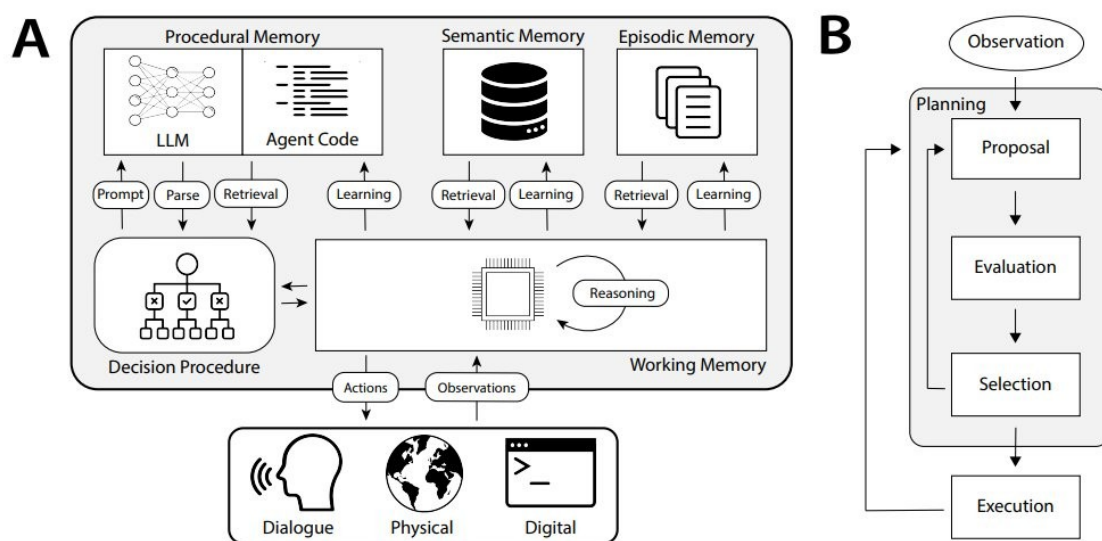
- * La mémoire sémantique renvoie à la connaissance de faits. Ce peut être des informations sur l'utilisateur et ses préférences, mais aussi des faits de culture générale, des informations sur des domaines de spécialité.
- * La mémoire dite épisodique renvoie aux expériences passées, c'est à dire en l'occurrence aux actions que l'agent a réalisées lors d'étapes précédentes du traitement. En effet, pour améliorer la

11 <https://arxiv.org/abs/2309.02427>

planification des actions à venir, l'agent doit être à tout instant conscient des actions déjà réalisées et de leur résultat.

* Enfin il y a la mémoire procédurale qui désigne les informations sur la façon dont l'agent fonctionne. Cela désigne donc sa mission globale, les outils qu'il doit effectuer, sa persona ... bref tout un ensemble d'informations que l'on fournit en général dans le system prompt du modèle.

L'argument de ce papier théorique est que, si l'agent pour bien mener sa mission à bout à besoin de toutes ces informations de nature différente, elles doivent, du fait de leur nature différente être stockées et récupérées de façon spécifique. Ils suggèrent dès lors une architecture bien plus complexe pour des agents dits « cognitifs », cognitive agents qui sont capables de manipuler dynamiquement plusieurs structures de mémoire différentes.



On fait donc finalement une distinction importante entre la mémoire à court terme (l'ensemble des informations données dans la fenêtre de contexte pour générer le message ou l'étape suivante) et la mémoire à long terme qui désigne l'ensemble des informations conservées à un moment sur l'exécution de l'agent. Mais ce travail reste toutefois théorique et ne propose pas d'implémentation de référence de ces «cognitive agents ». Dès lors, par quel(s) mécanisme(s) passer de la mémoire à long terme à la mémoire à court terme ?

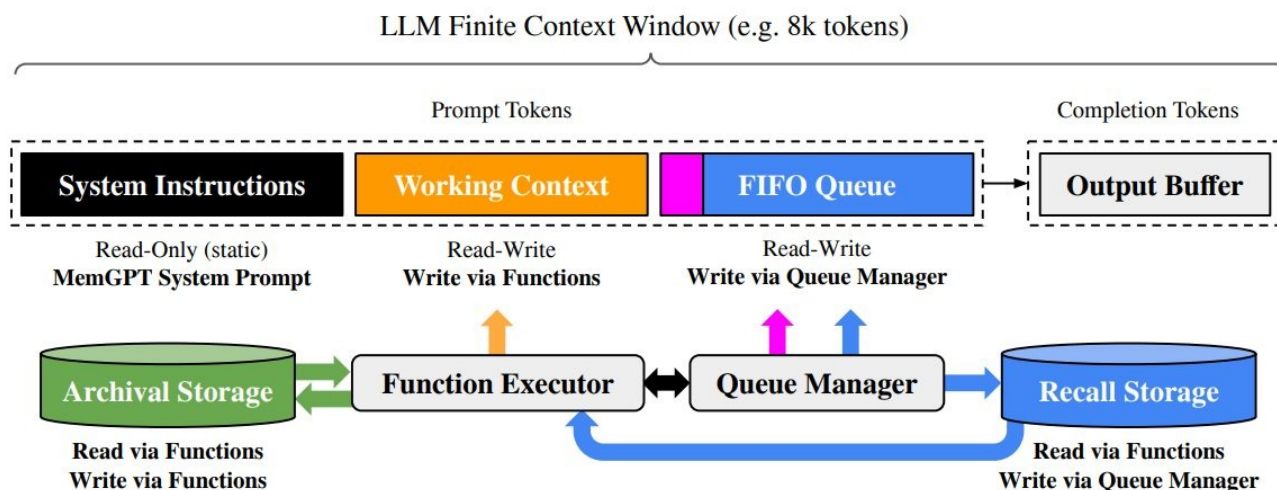
Un autre papier influent, MemGPT¹², propose quant à lui une implémentation de ces mécanismes de gestion de mémoire. La fenêtre de contexte du modèle pour la prochaine génération est remplie à partir :

- * du prompt système expliquant la fonction globale de l'agent
- * d'un certain nombre (queue FIFO) des derniers messages de la conversation
- * D'un certain nombre d'éléments essentiels sur l'utilisateur, ou sur l'agent et sa persona. Ces faits sont stockés dans une base de la forme clé valeur par des mécanismes d'édition de mémoire reposant sur le function calling. Les agents MemGPT ont la possibilité avec des outils prévus à cet effet de venir créer ou mettre à jour une information dans cette mémoire de faits essentiels (nommée

12 <https://arxiv.org/abs/2310.08560>

core memory). Ces informations sont systématiquement ajoutées au sein du system prompt lors de chaque génération.

* Ils peuvent aussi choisir de rechercher des informations sur les étapes d'exécution précédentes dans un index (nommé archival memory).



Ce papier est si je ne me trompe le premier à inclure la question de la mémoire au coeur de l'implémentation des agents IA alors que ce sujet était peu pris en compte dans les frameworks les plus utilisés. Ce mécanisme agentique d'édition et d'accès à la mémoire permet de gérer les limitations de la fenêtre de contexte mais de façon tributaire aux choix du LLM.

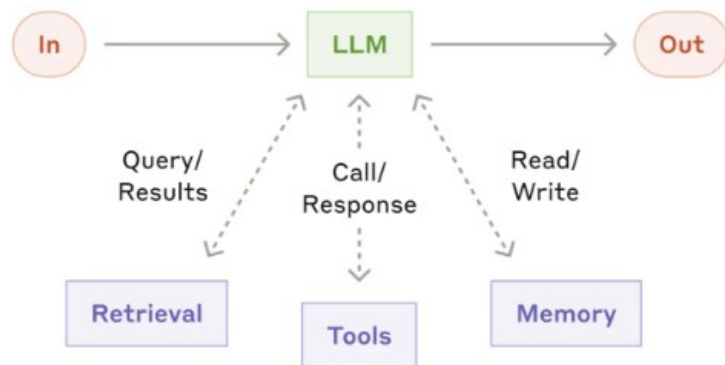
c) Unifier les différents composant : une question d'architecture logicielle

Les agents IA permettent d'automatiser des tâches complexes en connectant des LLMs (qui ont des facultés de raisonnement et de génération de contenu) à des ressources extérieures. Plusieurs approches sont ensuite possibles pour permettre d'automatiser des tâches :

* Soit prévoir une suite d'étapes précises avec des embranchements en utilisant les LLMs pour gérer certaines des tâches et effectuer certain des routages. Cela permet un grand contrôle sur le déroulement des tâches et limite les risques dus à des productions hallucinatoires de LLM en cours de traitement, mais limite la flexibilité du système. C'est cette approche qui a été privilégiée par Langchain.

* Soit on donne au LLM des capacités d'orchestration en le laissant décider à chaque fois des prochaines étapes à suivre sans guider précisément l'exécution. Cela permet plus de flexibilité mais apporte son lot de difficulté, notamment dans la gestion de l'exécution et l'évaluation des systèmes.

Construire des agents IA devient donc un véritable problème d'architecture logicielle où il faut trouver la meilleure manière de faire interagir un LLM avec les ressources extérieures. Les différentes bibliothèques proposant des agents se distinguent dans leur façon de concevoir et d'implémenter cette connexion. Plusieurs choses sont importantes à garder à l'esprit et restent valides pour tous les frameworks.

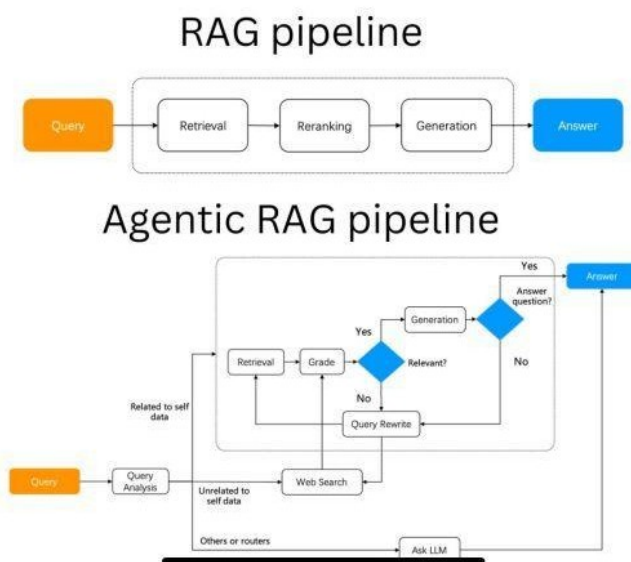


Le bloc central des systèmes agentiques reste le LLM augmenté (par une mémoire et des outils). Cette architecture peut observer et agir sur son environnement mais pas planifier son action. Elle ne permet « que » d'augmenter les capacités de réponse par rapport à un LLM simple. Toutefois ce simple apport de l'accès à des outils permet d'améliorer considérablement le potentiel applicatif dans des applications.

Par exemple, l'architecture RAG présentée dans le cours précédent est très utile pour compléter la réponse des LLMs avec des données extérieures et ainsi améliorer la fiabilité des réponses. Néanmoins elle souffre de limitations, car :

- * on a pas forcément besoin de ces données extérieures pour répondre à la question, et les documents récupérés pourraient apporter du bruit et dégrader la qualité de la réponse. La recherche est lancée de façon automatique qu'elle soit pertinente ou pas.
- * L'architecture RAG simple n'a pas de gestion de mémoire et donc ne permet pas les conversations suivies.
- * Il y a un manque de recul par rapport à la qualité de la réponse fournie. Le LLM va tenter coûte que coûte de répondre à partir des éléments fournis même si insuffisants sans évaluer la qualité de sa production.

Or utiliser la brique du « LLM augmenté » permettrait de palier à au moins deux de ces limitations :



* La gestion de la mémoire (optionnelle) est nativement intégrée

* La récupération de documents n'est pas systématiquement activée mais uniquement quand nécessaire ce qui permet un rendu plus naturel des échanges.

* L'activation ou non de la récupération de données est gérée entièrement par le mécanisme de function calling sans qu'il n'y ait de routage spécifique à implémenter

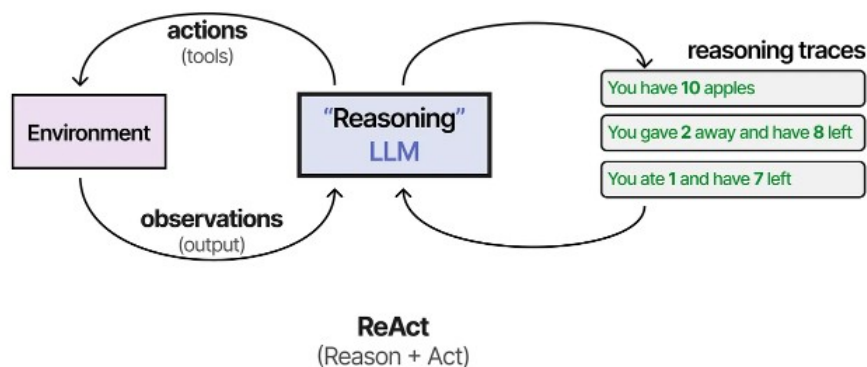
* On exploite les capacités de raisonnement du LLM qui optimise la requête pour trouver l'information (moins de dépendance à la formulation de départ de la question de l'utilisateur) et peut si besoin lancer plusieurs recherches successives pour trouver l'information (nécessaire pour répondre aux questions dites multi-hop).

* Optionnellement on peut rajouter une phase d'observation critique où la qualité de la réponse proposée est évaluée.

On obtient de ce fait un système de RAG agentique¹³ où le LLM décide quand rechercher des informations extérieures ou non, ce qui permet par exemple l'intégration plus facile du RAG dans des agents conversationnels, ou encore de connecter le LLM à plusieurs bases de données différentes en même temps. Le LLM augmenté devient le coeur de tout.

Néanmoins le RAG agentique a priori ne gère qu'un seul échange (génération d'une réponse à une question avec ou sans récupération de documents) mais certaines tâches demandent plus d'étapes. Par exemple pour la génération de code il pourrait être utile de faire en sorte que le LLM « relise » le code produit lors du premier jet, l'édite, ... et de même pour la génération de documents. C'est pourquoi il convient d'intégrer les LLMs augmentés dans des architecture plus larges. De nombreux travaux sont en cours actuellement.

L'architecture qui parmi toutes a obtenu le plus grand consensus est l'architecture React (reason + Act)¹⁴ qui n'est qu'une implémentation en boucle¹⁵ du LLM augmenté. Une fois qu'un appel d'outil ou une génération simple a été effectuée, une observation (thought) est ajoutée à la liste des messages générés de sorte que le modèle l'utilise pour décider de la prochaine étape à suivre, où d'arrêter si la réponse est satisfaisante.

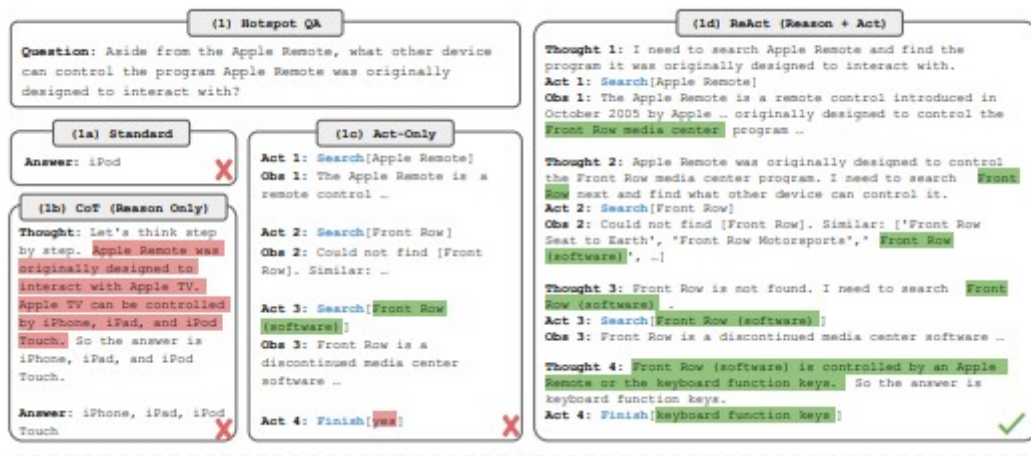


Le LLM a donc une faculté de communication. Ce type d'architecture multi step obtient sur de nombreuses tâches des résultats bien supérieur mêmes aux méthodes raffinées de prompt engineering. C'est pourquoi l'architecture ReACT est à la base de nombreux frameworks d'agents.

13 <https://community.ibm.com/community/user/blogs/armand-ruiz-gabernet/2025/01/15/forget-rag-and-welcome-agentic-rag?hlmlt=BL>

14 <https://arxiv.org/pdf/2210.03629>

15 <https://newsletter.maartengrootendorst.com/p/a-visual-guide-to-llm-agents> pour l'illustration



Ce n'est toutefois pas la seule architecture existante.

On peut classer les architectures d'agent selon leur fonctionnement interne. Par exemple smolagent de Hugging Face distingue les CodeAgents qui fonctionnent par génération de code python exécutable, des ToolCallingAgents qui exploitent le function calling et la sortie structurée en json. On peut aussi les classer selon le rôle des composants (routeur, optimiseur...) et au meilleur de ma connaissance actuelle aucune ressource ne répertorie tous les choix possibles. Il y a presque autant d'architecture que d'implémentations.

L'architecture ReAct est puissante mais limitée, car le LLM est limité en capacité par les outils qu'on lui donne, et des travaux ont prouvé qu'au delà d'une dizaine d'outils fournis, le choix fait n'était plus pertinent. On ne peut donc pas avec cette simple architecture donner aux LLM un accès illimité aux connaissances du monde ni un pouvoir illimité d'interaction, l'agent doit être spécialisé à une tâche.

Pour faire augmenter la complexité de ses agents, deux possibilités existent ;

- * ajouter plus d'outils et améliorer leur capacité à sélectionner le bon outil. Des tentatives sont en cours en créant des mécanismes de récupération d'outils à partir de la tâche à effectuer.
- * Complexifier les architectures pour utiliser au mieux le pouvoir de raisonnement.

Cette complexification peut se faire à partir de plusieurs constats :

- * Vu les limitations des agents isolés, on a pensé que la solution au problème serait de partir sur des architectures multi-agents. Il en existe de plusieurs types, certaines horizontales où des agents disposant d'outils différents collaborent à la même tâche et partagent certains éléments de leur état, ou d'autres architectures plus verticales où un agent planificateur est chargé de « manager » des agents spécialisés et leur délègue des missions plus précises.
- * L'architecture d'un agent IA est par définition modulaire. Un agent peut devenir l'outil d'un autre agent. Cela permet l'émergence d'architectures multi-agent. Il est très fréquent que des composants soient utilisés plusieurs fois.
- * Un agent conversationnel n'est rien d'autre qu'un agent qui intègre dans son architecture les entrées de l'utilisateur, que ce soit pour valider ce qui est proposé ou pour poursuivre la réflexion.
- * Cette complexification des architectures entraîne naturellement une augmentation de la complexité à contrôler et évaluer les agents, mais aussi une inflation coûteuse du nombre de LLMs à appeler.
- * Tout cela présuppose aussi que le LLM orchestre bien la suite d'actions à effectuer et propose un parcours censé. Mais comment s'assurer que la planification du LLM corresponde bien à la tâche ? C'est là où l'ingénierie des connaissances pourrait être utile (encore!).

III) Baser des agents IA sur des ontologies spécialisées : un champ de recherche à explorer.

Comme on l'a vu, pour automatiser des tâches complexes avec des LLMs il faut deux ingrédients : des outils spécifiques à la tâche qui permette au LLM de chercher des informations et effectuer des actions nécessaires à la tâche, et aussi de s'assurer d'une bonne planification pour une bonne exécution de la tâche. Cette phase est cruciale car de là dépend l'enchaînement des actions censée conduire à la réponse finale. Le plus souvent cette phase de planification est générique et s'appuie sur des appels de LLM puissants voire de LLM gérant des fonctionnalités de raisonnement (Open AI o1, Deepseek...) et n'est pas adaptée aux tâches spécifiques que l'on veut automatiser si elles sont liées à des domaines spécifiques. Les techniques d'ingénierie des connaissances peuvent donc s'avérer particulièrement pertinentes. Il est possible par exemple de

1) Connecter un agent à une base de connaissances (tool use)

L'ontologie ou le graphe de connaissance devient un outil activable si besoin par le LLM, ce qui serait l'équivalent d'un « agentic graph RAG ».

2) Exploiter les graphes pour structurer le workflow des agents

Les graphes peuvent être une représentation très adéquate pour représenter le flot d'exécution d'un agent IA. On voit alors clairement quel outil peut être utilisé, quelles sont les conditions entraînant certains routages...

Ce genre d'agents IA semblent être les remplaçants idéals des systèmes experts qui étaient créés à partir d'ontologies et de règles métier, apportant en plus de la souplesse d'exécution et plus de fonctionnalités.

Conserver la visualisation d'un workflow comme embranchement et structures conditionnelle est le parti pris par la librairie LangGraph qui est un des framework de création d'agents parmi les plus murs du marché.

Cette librairie permet l'orchestration de workflows qui intègrent des appels de LLM.

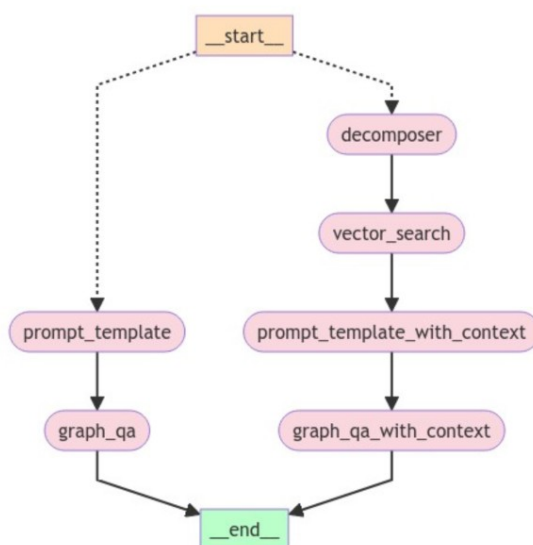
Dans LangGraph :

- * Chaque nœud est une fonction, que ce soit un appel à un LLM où une fonction qui s'exécute indépendamment (transformation de format de données, outil...)

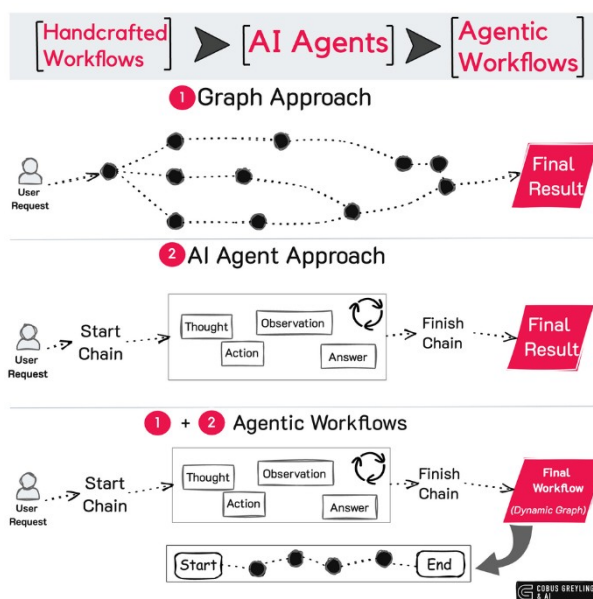
- * Certains nœuds sont des nœuds spéciaux dits de routage (routing) qui servent à aiguiller vers une des branches possibles.

- * Un edge n'est rien d'autre qu'un moyen de faire communiquer la sortie d'une fonction vers l'entrée d'une autre.

- * il faut donc bien faire attention au typage ...



Ainsi on peut facilement recoder et représenter des workflow tels que le graphRAG¹⁶ en condaant comme nœud toutes les fonctions dont on a besoin puis en créant un pipeline en ajoutant les edges.



Cela permet de pouvoir suivre les étapes d'exécution d'un exemple dans le graphe et d'avoir une traçabilité très claire des différentes étapes qui ont été suivies lors de l'exécution du workflow à partir d'un exemple spécifique. On peut limiter la prévisibilité du workflow en mettant en place des boucles (et ainsi recréer des architectures de type ReAct).

Même si une approche à base de graphe pure n'est pas adaptée à tous les cas d'usage, exploiter au moins la visualisation des architectures d'agents comme graphe est précieux pour l'explicabilité¹⁷.

Toutefois, ces graphes représentant des workflows adaptés à l'automatisation de tâches métier, de même que les systèmes experts du début de l'ingénierie des connaissances sont définis à la main par des ingénieurs et ne sont pas souples. Ils permettent d'automatiser des tâches spécifiques mais ne peuvent être créés sans une expertise humaine (allier des ingénieurs IA et plus des ingénieurs des connaissances) à des experts métier pour implémenter, tester et surveiller en production les agents). Finalement pas tant de progrès que ça en 30 ans... Il reste à trouver des solutions pour que la création et l'orchestration de ces workflows puisse se faire de façon automatique à partir de connaissances métier.

3) Exploiter les ontologies pour guider la planification des agents IA.

Une piste intéressante à exploiter serait l'intégration d'ontologies lors des phases de planification et d'orchestration des agents IA. Si par exemple dans le prompt système qui lance la planification pour exécuter une tâche spécifique, des fragments pertinents de l'ontologie étaient intégrés, cela serait susceptible d'augmenter à moindre coût la pertinence de la planification proposée. C'est une intuition qui semble assez évidente, mais qui reste peu explorée pour l'instant mis à part de rares articles ou projets industriels marginaux¹⁸.

¹⁶ <https://neo4j.com/blog/developer/neo4j-graphrag-workflow-langchain-langgraph/>

¹⁷ <https://cobusgreyling.medium.com/from-handcrafted-workflows-to-ai-agents-to-agentic-workflows-49e8f85daf7d>

¹⁸ <https://smythos.com/ai-agents/agent-architectures/agent-oriented-programming-and-ontologies/>

<https://scibite.com/knowledge-hub/news/what-is-agentic-ai-and-is-there-a-role-for-ontologies/>

<https://arxiv.org/abs/2405.17691>