

Les patterns de conception des systemes agents

Synthese en francais de l'ouvrage
Multi-Agent Systems with LLMs

Synthese basee sur
A. Gulli et al., *Multi-Agent Systems with LLMs*
(2025)

Mars 2026 — labo-llm.fr

Table des matieres

Introduction

1. Qu'est-ce qu'un agent IA ?

La boucle perception-planification-action

Quatre niveaux de sophistication

Le context engineering comme discipline fondatrice

2. Les fondamentaux de l'execution

Decomposition sequentielle (prompt chaining)

Routage conditionnel

Parallelisation et fire-and-consolidate

3. Boucles de feedback et auto-amelioration

Le modele Producer-Critic

Apprentissage et adaptation

Objectifs et monitoring

Techniques de raisonnement

4. Collaboration multi-agents et protocoles

Modeles d'interrelation

MCP : le protocole des outils

A2A : la communication inter-agents

Evaluer les systemes agents

5. Robustesse et securite

Gestion d'exceptions

L'humain dans la boucle

Optimisation des ressources

Guardrails multicouches

6. Ancrage, memoire et outils

L'usage d'outils par les agents

Planification adaptative

Architecture de la memoire

RAG et ancrage factuel

Priorisation dynamique

7. Perspectives et exploration

L'exploration des inconnues

Composition de patterns

Frameworks et ecosystemes

8. Huit enseignements accessibles

Glossaire des concepts cles

Sources et attribution

Introduction

Les systemes agents — ces programmes capables de percevoir leur environnement, de formuler un plan et d'agir de maniere autonome — representent l'une des evolutions les plus significatives dans le domaine de l'intelligence artificielle appliquee. Ils ne se contentent pas de generer du texte en reponse a une question : ils raisonnent, utilisent des outils, collaborent entre eux et apprennent de leurs erreurs.

L'ouvrage *Multi-Agent Systems with LLMs* d'A. Gulli et al. (2025) propose un inventaire methodique de 21 patterns de conception qui structurent ces systemes. Le livre s'adresse aux praticiens — ingenieurs, architectes logiciels, chefs de projet technique — qui construisent des systemes agents en conditions reelles. Il ne s'agit pas d'un traite theorique mais d'un guide pratique, illustre par des implementations concretes dans les principaux frameworks du marche (LangChain, LangGraph, Google ADK, CrewAI).

Ce document propose une synthese structuree de cet ouvrage, reorganisee par blocs thematiques pour faciliter la lecture continue. L'objectif est de rendre accessibles, en francais, les concepts fondamentaux et les patterns de conception que tout praticien des systemes agents devrait connaitre.

A propos de cette synthese. Ce document est une synthese analytique originale, pas une traduction. Il reorganise et reformule les contenus de l'ouvrage source pour un lectorat francophone technique. Les termes anglais etablis dans le domaine (prompt, chaining, tool use, guardrails, etc.) sont conserves tels quels.

1. Qu'est-ce qu'un agent IA ?

La boucle perception-planification-action

Un agent IA se distingue d'un chatbot ou d'un LLM brut par une caractéristique fondamentale : il opère en boucle. Là où un LLM standard reçoit une question et produit une réponse, un agent perçoit son environnement, formule un plan d'action, exécute ce plan via des outils externes, observe le résultat, et recommence si nécessaire. Cette boucle itérative est ce qui transforme un générateur de texte en système capable d'action autonome.

Gulli formalise cette distinction à travers la boucle perception-planification-action. L'agent reçoit des informations de son environnement (un fichier, une API, un retour utilisateur), raisonne sur la meilleure stratégie à adopter, exécute une ou plusieurs actions, puis évalue le résultat pour décider s'il a atteint son objectif ou s'il doit ajuster son approche.

Quatre niveaux de sophistication

Tous les agents ne sont pas égaux. L'ouvrage propose une taxonomie à quatre niveaux qui clarifie le spectre des systèmes agents :

- **Niveau 0 — Raisonnement seul.** Le modèle répond à partir de ses connaissances internes, sans outil externe. C'est le fonctionnement standard d'un LLM.
- **Niveau 1 — Outils externes.** L'agent peut invoquer des fonctions, interroger des API, exécuter du code. Il dépasse la génération de texte pour agir sur son environnement.
- **Niveau 2 — Context engineering stratégique.** L'agent ne se contente pas d'utiliser des outils : il construit délibérément le contexte informationnel qui guidera ses décisions. Il sélectionne, filtre et organise l'information avant de générer.
- **Niveau 3 — Multi-agents collaboratifs.** Plusieurs agents spécialisés collaborent, chacun avec son rôle, ses outils et son domaine d'expertise. La valeur émerge de leur coordination.

Le context engineering comme discipline fondatrice

L'un des apports les plus structurants de l'ouvrage est la formalisation du context engineering comme discipline a part entiere. L'idee centrale est simple mais puissante : la performance d'un agent ne depend pas principalement de la taille ou de la puissance du modele sous-jacent, mais de la qualite du contexte qu'on lui fournit.

Le context engineering consiste a construire un « tableau operationnel complet » pour l'agent avant qu'il ne genere le moindre token. Le critere d'inclusion est pragmatique : une information merite d'etre dans le contexte si, et seulement si, elle modifie le comportement de l'agent. Tout le reste est du bruit qui dilue l'attention.

Cette discipline s'applique a tous les niveaux : du prompt simple (quel contexte donner au LLM ?) a l'orchestration multi-agents (quelles informations chaque agent doit-il recevoir pour faire son travail ?).

2. Les fondamentaux de l'execution

Les trois operations elementaires que tout systeme agent doit maitriser sont la decomposition sequentielle, le routage conditionnel et la parallelisation. Ces patterns fondamentaux constituent la base sur laquelle tous les autres s'appuient.

Decomposition sequentielle (prompt chaining)

Le prompt chaining est le pattern le plus intuitif : plutot que de confier une tache complexe a un seul appel LLM, on la decompose en sous-problemes enchaines. Chaque etape recoit le resultat de la precedente comme entree et produit un resultat structure que la suivante pourra consommer.

La fiabilite de cette approche repose sur deux piliers. Le premier est l'utilisation de formats structures (JSON, XML) entre les etapes, pour que chaque maillon puisse parser sans ambiguite la sortie du precedent. Le second est l'attribution de roles distincts a chaque maillon : un agent qui a un role precis et un contexte cible performe bien mieux qu'un agent generaliste qui tenterait de tout faire en une seule passe.

Routage conditionnel

Le routage transforme un executeur statique en systeme adaptatif. Plutot que de suivre un chemin unique, le systeme evalue l'entree (ou l'etat courant) et achemine le flux vers le sous-processus le plus adapte.

Gulli identifie quatre mecanismes de routage :

- **Routage par LLM** : le modele decide lui-meme quel chemin emprunter, en se basant sur sa comprehension du contexte.
- **Routage par embeddings** : l'entree est comparee vectoriellement a des exemples de reference pour determiner la categorie.
- **Routage par regles** : des conditions explicites (if/else, regex, seuils) determinent le chemin.
- **Routage par classifieur ML** : un modele de classification entraine sur des donnees historiques effectue le routage.

Une distinction importante emerge entre le routage sur l'input present (le message entrant seul determine le chemin) et le routage sur l'etat accumule, ou l'historique complet de la conversation influence la decision. Ce second paradigme, porte notamment par LangGraph, ouvre la voie a des systemes veritablement adaptatifs.

Parallelisation et fire-and-consolidate

L'execution simultanee de taches independantes reduit la latence de maniere significative. Le pattern dominant en multi-agents est le fire-and-consolidate : on lance N branches paralleles, chacune traitant une portion independante du probleme, on attend la convergence de toutes les branches, puis on consolide les resultats en une synthese coherente.

Ce pattern est puissant mais exige de la rigueur dans sa mise en oeuvre. Il faut s'assurer que les branches sont reellement independantes (pas de dependance cachee entre elles), definir des criteres de convergence clairs (que faire si une branche echoue ou prend trop de temps ?), et concevoir une etape de consolidation qui ne se contente pas de concatener les resultats mais produit une synthese intelligente.

Gulli souligne un point technique important : en Python, la concurrence via asyncio masque la latence d'entree-sortie (attente de reponses API, lectures disque) mais n'offre pas de veritable parallelisme pour les taches gourmandes en calcul CPU. Cette distinction est cruciale pour les architectes de systemes agents.

3. Boucles de feedback et auto-amelioration

Les patterns de feedback sont ce qui permet aux agents de dépasser l'exécution lineaire pour entrer dans une logique d'amelioration iterative. Quatre dimensions structurent ce bloc : la reflexion, l'apprentissage, le suivi d'objectifs et les techniques de raisonnement avance.

Le modele Producer-Critic

Le pattern reflexion repose sur une idee simple mais efficace : separer la generation de l'evaluation. Un agent « producteur » genere un resultat, puis un agent « critique » l'evalue selon des criteres explicites. Le producteur revise alors sa sortie en fonction des critiques, et le cycle recommence jusqu'a satisfaction des criteres.

Cette separation est essentielle pour eliminer les biais d'auto-revision. Quand le meme agent genere et evalue son propre travail, il a tendance a trouver sa production satisfaisante — un biais bien documente. En confiant l'evaluation a un agent distinct (ou au meme modele mais avec un prompt d'evaluation specifique), on introduit une forme de contradiction productive.

Le risque principal est la boucle infinie : sans criteres d'arret explicites et mesurables, le cycle producteur-critique peut tourner indefiniment. Il est imperatif de definir des conditions d'arret claires (nombre maximum d'iterations, seuil de qualite atteint, temps ecoule).

Apprentissage et adaptation

Les agents ne sont pas condammes a rester statiques. Gulli presente trois modalites d'evolution :

- **Apprentissage par renforcement** (PPO, DPO) : l'agent optimise son comportement en fonction de recompenses explicites.
- **Apprentissage supervise** : l'agent s'ameliore a partir d'exemples corriges par des humains.
- **Decouverte autonome** : l'agent explore et decouvre de nouvelles strategies sans supervision directe.

Deux systemes illustrent ces approches de maniere remarquable. SICA (Self-Improving Coding Agent) est un agent de programmation qui modifie son propre code de maniere iterative, conservant les strategies qui

fonctionnent et abandonnant celles qui échouent. AlphaEvolve, développé par Google, va plus loin en découvrant de nouveaux algorithmes par évolution — une forme d'exploration créative guidée par des critères de performance.

Objectifs et monitoring

Le pattern Goal Setting transforme les agents réactifs (qui répondent à des stimuli) en entités orientées but (qui poursuivent un objectif). L'approche s'appuie sur trois piliers : la formulation SMART des objectifs (spécifiques, mesurables, atteignables, pertinents, temporellement définis), une boucle d'auto-évaluation continue, et des critères d'arrêt mesurables qui permettent à l'agent de déterminer s'il a atteint son but.

Techniques de raisonnement

Les techniques de raisonnement structurent la façon dont un agent « réfléchit » avant d'agir. Quatre approches majeures coexistent :

Chain-of-Thought (CoT) force le modèle à expliciter ses étapes de raisonnement intermédiaires. Plutôt que de sauter directement à la conclusion, l'agent décompose le problème en étapes logiques enchaînées. Cette décomposition linéaire améliore significativement la performance sur les tâches de raisonnement complexe.

Tree-of-Thought (ToT) généralise le CoT en explorant plusieurs chemins de raisonnement simultanément. L'agent construit un arbre de possibilités, évalue chaque branche, et peut revenir en arrière si un chemin s'avère improductif. C'est l'équivalent d'un joueur d'échecs qui évalue plusieurs coups avant de choisir.

ReAct (Reasoning + Acting) entrelace les phases de raisonnement et d'action. L'agent alterne entre réflexion (« je dois trouver X »), action (interroger une API, lire un fichier) et observation (analyser le résultat). Ce cycle Thought-Action-Observation est le pattern dominant pour les agents autonomes.

Scaling Inference Law. Un résultat contre-intuitif mais bien documenté : la performance est proportionnelle aux ressources computationnelles allouées à l'inférence, pas à la taille du modèle. Concrètement, un modèle plus petit qui « réfléchit plus longtemps » (davantage de tokens de raisonnement) peut surpasser un modèle plus grand qui répond directement.

Des techniques plus avancées comme les Chain of Debates (CoD) et Graph of Debates (GoD) structurent le consensus entre plusieurs agents par débat contradictoire. MASS (Multi-Agent System Search) optimise automatiquement les prompts par bloc puis la topologie du workflow, une forme de méta-optimisation des systèmes agents.

4. Collaboration multi-agents et protocoles

Les systemes multi-agents representent le niveau de sophistication le plus eleve dans la taxonomie de Gulli. Leur efficacite repose sur des modeles d'organisation bien definis et des protocoles de communication standardises.

Modeles d'interrelation

Six modeles structurent les relations entre agents :

Modele	Description	Cas d'usage
Agent simple	Un seul agent avec ses outils	Taches unitaires, prototypage
Reseau pair-a-pair	Agents egaux communiquant librement	Brainstorming, exploration
Superviseur	Un agent coordonne les autres	Workflow structures
Superviseur-outil	Les sous-agents sont exposes comme outils du superviseur	Orchestration fine
Hierarchique	Plusieurs niveaux de supervision	Systemes complexes, grande echelle
Custom	Topologie sur mesure	Architectures specifiques

Le modele superviseur domine les implementations en production. Un agent orchestrateur decompose la tache, delegue aux agents specialises, collecte les resultats et consolide. Le modele hierarchique ajoute des niveaux intermediaires pour les systemes de grande echelle, ou un seul superviseur ne suffit plus a coordonner l'ensemble.

L'efficacite de la collaboration depend de deux facteurs critiques : des protocoles de communication standardises (pour que les agents se comprennent) et une ontologie partagee (un vocabulaire commun pour decire les taches, les resultats et les etats).

MCP : le protocole des outils

Le Model Context Protocol (MCP) est un standard ouvert qui formalise la relation client-serveur entre un agent et ses outils. L'agent (client) decouvre dynamiquement les outils disponibles (serveurs), interroge leurs capacites, et les invoque de maniere standardisee.

Gulli souligne une distinction critique entre deux approches : le « wrapper MCP sur API legacy » (on enveloppe une API existante dans le protocole MCP, souvent avec des resultats mediocres) et l' « API optimisee pour agents » (l'API est concue des le depart pour etre consommee par des agents, avec filtrage intelligent, pagination adaptee et format Markdown lisible).

Le benefice fondamental de MCP est le decouplage intelligence/capacites : l'agent n'a pas besoin de connaitre les details d'implementation de chaque outil. Il decouvre ce qui est disponible, comprend comment l'utiliser, et l'invoque via un protocole uniforme. Cela permet une composabilite reelle : ajouter un nouvel outil ne necessite pas de modifier l'agent.

A2A : la communication inter-agents

Si MCP standardise la relation agent-outil, le protocole Agent-to-Agent (A2A) standardise la communication entre agents eux-memes. Base sur HTTP et RPC, A2A permet a des agents construits avec des frameworks differents de collaborer.

Trois elements structurent A2A :

- **Agent Card** : un fichier JSON declarant l'identite de l'agent, ses capacites, ses endpoints et ses modes d'interaction.
- **Quatre modes d'interaction** : synchrone (requete-reponse immediate), asynchrone par polling (l'appelant verifie periodiquement le statut), Server-Sent Events (flux continu), et webhooks (notification a completion).
- **Gestion de taches** : chaque interaction est modelisee comme une tache avec des etats explicites (soumise, en cours, terminee, echouee), permettant le suivi et la reprise sur erreur.

A2A resout un probleme fondamental : l'isolation des agents individuels. Sans standard de communication, chaque paire d'agents necessite un connecteur specifique. Avec A2A, tout agent conforme peut communiquer avec tout autre agent conforme, ouvrant la voie a des ecosystemes veritablement composables.

Evaluer les systemes agents

L'evaluation des systemes agents differe fondamentalement du test logiciel classique. On ne se contente pas de verifier qu'une sortie est correcte : on evalue la trajectoire complete de l'agent — le chemin qu'il a emprunte pour arriver au resultat.

Trois approches structurent l'evaluation :

- **Metriques de trajectoire** : comparaison entre le chemin reel emprunte par l'agent et le chemin ideal. Un agent qui arrive au bon resultat par un chemin aberrant (trop d'etapes, outils inutiles, boucles) revele un probleme d'architecture.
- **LLM-as-a-Judge** : un LLM evalue la qualite des sorties selon une grille de criteres structures. Cette approche permet de dimensionner l'evaluation sans recourir systematiquement a des evaluateurs humains.
- **Monitoring continu** : surveillance en production de la derive des performances dans le temps. Un systeme agent qui fonctionne bien aujourd'hui peut degrader progressivement si les donnees ou les APIs qu'il utilise evoluent.

Le modele « Contractor » propose une evolution ambitieuse : plutot que de donner des instructions vagues a un agent, on redige une specification formelle (comme un contrat de prestation), avec des criteres de succes mesurables, des conditions de negociation, et des mecanismes d'auto-validation. L'agent s'engage sur un resultat, pas sur un effort.

5. Robustesse et securite

Un systeme agent qui fonctionne en conditions ideales ne suffit pas. Les systemes deployes en production doivent gerer les erreurs, collaborer avec les humains, optimiser leurs ressources et resister aux usages malveillants.

Gestion d'exceptions

Gulli propose une architecture de gestion d'erreurs a trois strates :

- **Detection** : identifier qu'une erreur s'est produite (retour d'API inattendu, timeout, format invalide).
- **Traitement tactique** : appliquer la reponse appropriee selon le type d'erreur. Les erreurs retryable (timeout reseau, rate limit) justifient une nouvelle tentative. Les erreurs permanentes (API inexistante, permission refusee) necessitent un chemin alternatif.
- **Stabilisation** : maintenir un fonctionnement degrade plutot que de s'effondrer completement. Le principe de graceful degradation garantit que le systeme continue a fournir une valeur partielle meme quand certains composants echouent.

Deux patterns de fallback meritent attention : le sequential agent fallback (essayer un agent, puis un autre, puis un autre) et le state-driven fallback (le choix de l'agent de remplacement depend de l'etat courant du systeme). Le couplage entre la reflexion et la gestion d'exceptions permet l'auto-correction : l'agent analyse l'erreur, identifie la cause probable, et ajuste sa strategie.

L'humain dans la boucle

La complementarite humain-agent est ce qui rend les systemes agents deployables en conditions reelles. Les agents excellent sur le volume et la vitesse ; les humains apportent le jugement contextuel et la dimension ethique. Cette complementarite fonde la legitimite du deploiement.

Deux paradigmes coexistent :

- **Human-in-the-loop** : l'humain supervise chaque etape significative. L'agent propose, l'humain valide. Ce mode convient aux taches a forte criticite ou l'erreur est couteuse.

- **Human-on-the-loop** : l'humain definit la politique et les garde-fous, puis l'agent execute de maniere autonome, rapportant a completion. Ce mode convient aux taches repetitives ou la politique est bien definie et les risques maitrises.

L'evolution naturelle va du premier vers le second : a mesure que la confiance s'etablit et que les garde-fous se renforcent, l'autonomie de l'agent augmente. La boucle de feedback RLHF (Reinforcement Learning from Human Feedback) permet cette transition progressive : les corrections humaines alimentent l'amelioration continue de l'agent.

Optimisation des ressources

Tous les appels ne justifient pas le meme modele. Le routage dynamique entre modeles constitue l'un des leviers d'optimisation les plus efficaces : utiliser un modele leger et rapide (type Flash ou Haiku) pour les taches simples (classification, extraction), et un modele puissant (type Pro ou Opus) pour les taches complexes (analyse, synthese, raisonnement).

Ce dispatch par type cognitif — router non pas selon la longueur de la tache mais selon le type de cognition requis — produit un rapport cout/performance nettement superieur a l'approche « un seul modele pour tout ». Le pruning contextuel (n'inclure dans le contexte que l'information qui modifie le comportement de l'agent) reduit la consommation de tokens sans degrader la qualite.

Un agent critique peut surveiller les routages et detecter les choix suboptimaux (un modele couteux utilise pour une tache triviale), permettant une amelioration continue de la politique de routage.

Guardrails multicouches

Les guardrails ne sont pas une contrainte sur les capacites de l'agent : ils sont ce qui rend ses capacites deployables. Gulli propose une defense en profondeur a cinq couches :

1. **Validation des entrees** : verifier que les donnees recues sont conformes au format attendu et ne contiennent pas d'injection malveillante.
2. **Filtrage des sorties** : verifier que les reponses generees respectent les contraintes de contenu (pas d'information sensible, pas de contenu inapproprié).
3. **Contraintes comportementales** : limiter ce que l'agent peut faire (interdiction d'accéder a certaines ressources, plafond sur le nombre d'actions).
4. **Restrictions d'outils** : appliquer le principe du moindre privilege — chaque agent n'a acces qu'aux outils strictement necessaires a sa mission.

5. **Checkpoint et rollback** : sauvegarder l'etat du systeme a intervalles reguliers pour pouvoir revenir en arriere en cas de derive.

La validation structuree (par exemple via Pydantic en Python) automatise la verification de conformite des outputs : chaque sortie d'agent est validee contre un schema avant d'etre transmise au consommateur suivant. Cette approche transforme les erreurs silencieuses en erreurs detectees et traitees.

6. Ancrage, memoire et outils

Ce bloc couvre les mecanismes par lesquels les agents se connectent au monde reel (outils), conservent leur experience (memoire), et s'ancrent dans des faits verifiables (RAG).

L'usage d'outils par les agents

Le tool use est le mecanisme qui transforme un generateur de texte en systeme capable d'action. Le cycle complet se deroule en six etapes :

1. **Definition** : l'outil est decrit au modele (nom, parametres, comportement attendu).
2. **Decision** : le LLM determine s'il doit utiliser un outil, et lequel.
3. **Generation** : le modele genere l'appel d'outil avec les parametres appropries.
4. **Execution** : l'outil est effectivement execute (appel API, requete base de donnees, execution de code).
5. **Observation** : le resultat de l'execution est renvoye au modele.
6. **Traitement** : le modele integre le resultat et decide de la suite (repondre, utiliser un autre outil, iterer).

Le concept de tool use s'etend bien au-dela des simples fonctions : API externes, bases de donnees, et meme delegation a d'autres agents. Un agent peut « utiliser » un autre agent comme outil, brouillant la frontiere entre outil et collaborateur.

Planification adaptative

Le pattern plan-execute-adapt est le coeur de l'intelligence agentic. L'agent formule un plan, execute les premieres etapes, evalue les resultats, puis revise le plan si necessaire. Cette capacite d'adaptation distingue fondamentalement un agent d'un workflow fixe.

Gulli trace une distinction importante entre la planification dynamique (le chemin est a decouvrir — on ne sait pas a l'avance combien d'etapes seront necessaires ni lesquelles) et le workflow fixe (le chemin est connu a l'avance, l'execution est deterministe). Les deux ont leur place, mais les systemes agents

excellents combinent les deux : un workflow fixe pour la structure globale, avec des points de planification dynamique pour les étapes qui nécessitent de l'adaptation.

Google DeepResearch illustre cette approche avec brio : l'agent formule un plan de recherche, exécute les premières requêtes, identifie les lacunes dans les résultats, replanifie pour combler ces lacunes, et itère jusqu'à avoir couvert le sujet de manière satisfaisante.

Architecture de la mémoire

La mémoire des agents s'organise selon une architecture duale :

Mémoire à court terme (short-term) : la fenêtre de contexte du modèle et l'historique de la session courante. Cette mémoire est immédiate, précise, mais limitée en taille et éphémère.

Mémoire à long terme (long-term) : les bases de données vectorielles, les graphes de connaissances, les services de mémoire persistants. Cette mémoire survit aux sessions, supporte de grands volumes, mais nécessite des mécanismes de récupération efficaces.

Gulli distingue trois types de mémoire fonctionnels :

- **Mémoire sémantique** : les faits et connaissances (« Python est un langage de programmation »).
- **Mémoire épisodique** : les exemples et expériences passés (« la dernière fois que j'ai traité ce type de requête, j'ai procédé ainsi »).
- **Mémoire procédurale** : les instructions et règles de comportement (« quand je reçois une requête de type X, je dois d'abord vérifier Y »). Fait remarquable : ces instructions sont modifiables, ce qui signifie qu'un agent peut, via le pattern réflexion, modifier son propre prompt.

RAG et ancrage factuel

Le Retrieval-Augmented Generation (RAG) est le mécanisme par lequel un agent ancre ses réponses dans des données externes vérifiables, plutôt que de s'appuyer uniquement sur les connaissances encodées dans ses poids.

Gulli identifie trois niveaux de sophistication croissante :

- **RAG standard** : recherche vectorielle passive. On cherche les documents les plus proches sémantiquement de la requête, on les injecte dans le contexte, et le modèle génère sa réponse. Simple mais limité (pas de vérification de pertinence, pas de réconciliation de sources contradictoires).

- **GraphRAG** : navigation de relations structurees. Au lieu de chercher des documents isoless, on navigue un graphe de connaissances qui encode les relations entre entites. Permet des raisonnements multi-sauts (« qui est le directeur de l'entreprise qui a acquis X ? »).
- **Agentic RAG** : le RAG augmente d'une couche de raisonnement. L'agent valide la pertinence des sources recuperees, reconcilie les informations contradictoires, decompose les requetes complexes en sous-requetes, et peut decider de chercher davantage si les premiers resultats sont insuffisants.

La recherche hybride (combinant BM25 pour la recherche lexicale, vecteurs pour la recherche semantique, et metadonnees pour le filtrage) offre la robustesse maximale. Aucune methode seule ne couvre tous les cas — leur combinaison compense les faiblesses de chacune.

Priorisation dynamique

La priorisation est ce qui transforme un agent executant mecaniquement une liste de taches en agent veritablement strategique. Gulli propose cinq dimensions evaluatives : urgence, importance, dependances, ressources disponibles, et ratio cout-benefice.

Ces dimensions s'appliquent a trois niveaux :

- **Strategique** : quels objectifs poursuivre en priorite.
- **Tactique** : quel plan adopter pour atteindre l'objectif choisi.
- **Operationnel** : dans quel ordre executer les etapes du plan.

La signature de l'agenticite veritable, selon Gulli, est la re-priorisation dynamique en cours d'execution. Un agent qui decouvre, en milieu de tache, qu'une de ses hypotheses est fausse ou qu'une nouvelle information change la donne, doit etre capable de reordonner ses priorites sans intervention humaine. C'est la difference entre un script et un agent.

7. Perspectives et exploration

L'exploration des inconnues

L'un des chapitres les plus prospectifs de l'ouvrage traite de l'exploration — la capacité à dépasser l'optimisation (faire mieux ce qu'on sait faire) pour identifier les « unknown unknowns » (découvrir ce qu'on ne savait pas ne pas savoir).

Deux systèmes illustrent cette ambition. Google Co-Scientist déploie six agents spécialisés (génération d'hypothèses, réflexion, classement par score Elo, évolution) pour explorer des espaces de recherche scientifique. Agent Laboratory implémente un workflow de recherche complet — de la revue de littérature à la rédaction de l'article — chaque étape étant prise en charge par un agent dédié.

Ces systèmes restent expérimentaux, mais ils pointent vers un usage des agents qui dépasse l'automatisation de tâches connues : l'assistance à la découverte et à la créativité structurée.

Composition de patterns

La conclusion de l'ouvrage revient sur une idée centrale : les 21 patterns ne sont pas isolés. Ils se combinent, se renforcent mutuellement, et c'est de leur composition que naît la valeur des systèmes agents.

Les patterns se regroupent en quatre catégories :

- **Execution et décomposition** : chaining, routage, parallélisation, planification.
- **Interaction avec l'environnement** : tool use, RAG, MCP, mémoire.
- **Apprentissage et auto-amélioration** : réflexion, apprentissage, objectifs, raisonnement.
- **Collaboration et communication** : multi-agents, A2A, HITL, évaluation.

Un système agent en production combine typiquement des patterns de chaque catégorie. Un workflow de recherche documentaire, par exemple, peut combiner la planification adaptative (pour structurer la recherche), le RAG (pour récupérer les sources), la réflexion (pour évaluer la qualité), et le multi-agents (pour paralléliser les recherches sur différentes facettes du sujet).

La trajectoire anticipee par Gulli va du human-in-the-loop (supervision humaine constante) vers le human-on-the-loop (politique humaine, execution agent), soutenue par des ecosystemes standardises (MCP, A2A) et une integration croissante de techniques neuro-symboliques.

Frameworks et ecosystemes

L'ouvrage propose un panorama detaille des frameworks disponibles, organise par niveau d'abstraction :

Niveau	Framework	Paradigme
Bas niveau	LangChain	Graphes diriges acycliques (DAG), chaines lineaires
Intermediaire	LangGraph	Graphes stateful avec boucles et etats accumules
Haut niveau	CrewAI, Google ADK	Orchestration d'equipes, abstraction roles/taches

LangGraph represente une avancee notable par rapport a LangChain en introduisant les boucles (un noeud peut revisiter un etat precedent) et les etats accumules (chaque noeud a acces a l'historique complet des transitions). Ces deux capacites sont essentielles pour implementer les patterns adaptatifs decrits dans l'ouvrage.

Les orchestrateurs de haut niveau (CrewAI, Google ADK) ajoutent une couche d'abstraction supplementaire : on definit des agents avec des roles, des objectifs et des outils, puis on les organise en equipes avec des protocoles de collaboration. La contrepartie est une perte de controle fin sur les interactions.

8. Huit enseignements accessibles

Pour conclure, voici huit idées-forces extraites de l'ouvrage, formulées pour être immédiatement compréhensibles :

- 1. Un agent IA n'est pas un chatbot amélioré.** Un agent se distingue par sa capacité à percevoir son environnement, formuler un plan, agir via des outils externes et apprendre de l'expérience. Cette boucle itérative le place dans une catégorie fondamentalement différente du système question-réponse.
- 2. Diviser pour mieux résoudre : le prompt chaining.** Plutôt que de poser une question géante à un modèle, on décompose le problème en étapes simples dont chaque résultat nourrit l'étape suivante. La fiabilité vient de la modularité, pas de la taille du modèle.
- 3. Le contexte prime sur la puissance du modèle.** Ce n'est pas le modèle le plus gros qui gagne, mais celui qui reçoit les bonnes informations au bon moment. Un modèle modeste avec un contexte bien construit surpasse souvent un modèle massif mal informé.
- 4. Les équipes d'agents surpassent l'agent unique.** Les problèmes complexes se résolvent mieux par des équipes d'agents spécialisés (un chercheur, un rédacteur, un critique) que par un super-agent généraliste. La valeur émerge de la coordination, pas de l'omniscience.
- 5. L'auto-correction transforme la qualité.** Le pattern réflexion sépare la génération de l'évaluation : un agent produit, un autre critique. Cette séparation élimine les biais d'auto-complaisance et permet un affinage itératif mesurable.
- 6. Les protocoles ouverts changent la donne.** MCP (pour les outils) et A2A (pour la communication inter-agents) standardisent les échanges. Un agent construit avec un framework peut collaborer avec un agent d'un autre framework, ouvrant la voie à des écosystèmes composables.
- 7. La sécurité est une architecture, pas une option.** Les garde-rails multicouches (validation des entrées, filtrage des sorties, principe du moindre privilège, checkpoint/rollback) ne limitent pas les capacités de l'agent : ils les rendent déployables en conditions réelles.
- 8. Penser, pas seulement générer.** Les techniques de raisonnement (Chain-of-Thought, Tree-of-Thought, ReAct) forcent l'agent à expliciter ses étapes intermédiaires. Un modèle plus petit qui « réfléchit plus longtemps » peut surpasser un modèle plus grand qui répond directement.

Glossaire des concepts clés

Concept	Definition
Context Engineering	Discipline de construction d'un environnement informationnel complet pour l'agent avant generation de tokens. Critere : l'information change-t-elle le comportement de l'agent ?
Fire-and-consolidate	Pattern d'execution : N taches paralleles independantes, convergence, consolidation en synthese.
Routing (Routing)	Decision conditionnelle acheminant le flux vers le sous-processus le plus adapte.
Reflexion (Reflexion)	Boucle d'auto-evaluation et d'affinage iteratif des outputs.
Producer-Critic	Separation generation/evaluation pour eliminer les biais d'auto-revision.
Tool Use / Function Calling	Mecanisme par lequel l'agent decide d'invoquer un outil externe et traite le resultat.
Planification adaptative	Formulation d'un plan, execution iterative, revision dynamique face aux obstacles.
Memoire duale (STM/LTM)	Architecture court-terme (context window) + long-terme (vector DB, knowledge graph).
RAG	Retrieval-Augmented Generation : recuperation d'information externe avant generation pour ancrage factuel.
Agentic RAG	RAG augmente d'une couche de raisonnement : validation source, reconciliation conflits, decomposition requete.
MCP	Model Context Protocol : standard ouvert client-serveur pour decouverte et consommation dynamique d'outils.
A2A	Agent-to-Agent : protocole HTTP-RPC ouvert pour interoperabilite entre agents heterogenes.
	Technique de prompting forçant la generation d'etapes intermediaires de raisonnement.

Chain-of-Thought (CoT)	
Tree-of-Thought (ToT)	Exploration arborescente de chemins de raisonnement multiples avec retour arriere.
ReAct	Boucle interleaved Thought-Action-Observation pour agents autonomes.
Scaling Inference Law	Performance proportionnelle aux ressources computationnelles allouees a l'inference, non a la taille du modele.
Guardrails	Defense multicouche : validation, filtrage, contraintes, checkpoint/rollback.
LLM-as-a-Judge	Evaluation qualitative par un LLM avec rubrique structuree.
Graceful Degradation	Maintien de fonctionnalite partielle plutot que defaillance totale sous contrainte.
Human-in/on-the-Loop	In : supervision humaine a chaque pas. On : politique humaine, execution agent autonome.
Dispatch cognitif	Routage des sous-taches par type de cognition requise (extraction vers modele leger, analyse vers modele lourd).

Sources et attribution

Ce document est une synthèse analytique originale. Il ne constitue pas une traduction de l'ouvrage source.

Ouvrage source :

- Gulli, A. et al. (2025). *Multi-Agent Systems with LLMs*.

Concepts et systemes references dans la synthese :

- LangChain / LangGraph — frameworks open source pour applications LLM et agents (Harrison Chase et al.)
- CrewAI — framework d'orchestration multi-agents (Joao Moura et al.)
- Google ADK (Agent Development Kit) — framework de developpement agents Google
- AlphaEvolve — systeme de decouverte d'algorithmes par evolution (Google DeepMind)
- SICA (Self-Improving Coding Agent) — agent de programmation auto-ameliorant
- MCP (Model Context Protocol) — standard ouvert Anthropic
- A2A (Agent-to-Agent Protocol) — standard ouvert Google
- MASS (Multi-Agent System Search) — meta-optimisation de workflows agents
- Google Co-Scientist — systeme multi-agents de recherche scientifique

Synthese basee sur *Multi-Agent Systems with LLMs*, A. Gulli et al. (2025).

Publication : labo-llm.fr — Mars 2026.