Créez une équipe

de 20 personnes avec l'IA.

Livret avancé : concevoir des systèmes vivants

Philippe Anel

2026

Ce livret accompagne la Masterclass IA « Créer une équipe de 20 personnes avec l'IA », donnée le 6 avril 2026. Niveau avancé.

Pour toute remarque ou suggestion : `support@scanderia.com`

Table des matières

Chapitre 1

Pourquoi un système d'agents devient vivant

Vos agents tournent, vos fichiers se remplissent. Quelque chose manque quand même. Ce qui manque, dans mon expérience, c'est la décision. Ce chapitre pose la thèse du livret : un système d'agents vivant tient sur trois briques (état partagé, boucle décisionnelle, conversations traçables) dont la décision est le pivot.

Une remarque revient souvent dans les conversations que j'ai avec des constructeurs avancés : « Mon système marche, mais je ne sais pas s'il fait quoi que ce soit d'intéressant. » J'ai d'abord cru à une formule. C'est devenu, au fil des mois, le symptôme le plus fréquent que je rencontre.

Le système marche. Les agents s'exécutent. Les fichiers se remplissent. Quelque chose manque quand même.

Ce qui manque, dans mon expérience, c'est la décision.

Trois manières de faire travailler une équipe d'agents

Je reprends la progression de la collection, parce qu'elle correspond à trois maturités réelles que j'observe.

Le livret débutant montre **un agent qui exécute**. Vous lui donnez un dossier, il fait son travail. C'est déjà beaucoup : la plupart des gens n'en sont pas encore là et ce niveau résout la majeure partie des cas d'usage individuels (production de contenu, veille, analyse de documents). L'agent suit des instructions claires, sans décision à prendre.

Le livret intermédiaire montre **une équipe qui orchestre**. Plusieurs agents, un manifeste, des fichiers partagés en mode passif (un agent lit un journal écrit par un autre, le chef de bureau synthétise). Il y a une coordination, mais le scénario est câblé d'avance. Si demain l'environnement change (un client nouveau, une donnée inattendue, une priorité qui bouge), le système ne s'adapte pas tout seul. Il faut le reprogrammer.

C'est ce que j'appelle l'orchestration statique. Elle est solide et prévisible. Pour beaucoup de métiers, c'est exactement le bon niveau. Pour d'autres, ce n'est pas suffisant.

Le livret avancé attaque le niveau au-dessus, ce que j'appelle l'orchestration dynamique : **une équipe d'agents qui décide**. Chaque agent regarde un état partagé, choisit son prochain coup en fonction de ce qu'il voit. Il laisse une trace de sa décision pour les autres. Ce n'est plus un

workflow câblé d'avance, c'est une boucle qui s'adapte.

Trois maturités, trois niveaux de complexité technique, trois publics différents.

Pourquoi je préfère parler de « vivant » plutôt que d'« intelligent »

Le mot « intelligent » est piégé. Il évoque une autonomie qu'on ne sait pas encore garantir et il déclenche chez le client soit l'enthousiasme naïf soit la peur réflexe. Aucun des deux ne sert.

J'utilise « vivant » pour désigner quelque chose de plus concret : un système dont le comportement ne peut pas être prédit en lisant le code, parce qu'il dépend de l'état dans lequel il se trouve à un instant donné.

Un cron qui lance un agent toutes les heures n'est pas vivant. Vous savez exactement ce qu'il va faire.

Un système qui regarde l'état de sa file de tâches, voit qu'une tâche est urgente, replanifie son ordre d'exécution et notifie l'humain : ce système est vivant. Pas parce qu'il est intelligent (il suit des règles), mais parce que sa trajectoire dépend d'un contexte qu'il découvre en chemin.

C'est cette qualité que ce livret cherche à enseigner.

Trois briques

Pour qu'un système d'agents devienne vivant, il faut trois choses et elles se tiennent les unes les autres.

Un état partagé. Quelque chose que tous les agents peuvent lire et modifier de manière cohérente. Ça peut être un fichier, mais souvent c'est insuffisant : deux agents qui écrivent au même moment se marchent dessus, l'historique est difficile à reconstituer, la concurrence est mal gérée. On verra dans le chapitre 2 quand le markdown ne suffit plus et par où passer ensuite.

Une boucle décisionnelle. Chaque agent doit pouvoir choisir son prochain coup en fonction de l'état qu'il observe. Pas une autonomie aveugle ; une décision encadrée, avec des règles explicites, des garde-fous et une remontée vers l'humain quand on sort du périmètre. C'est le sujet du chapitre 3.

Des conversations traçables. Quand un agent décide quelque chose, les autres doivent pouvoir comprendre pourquoi. Les conversations entre agents (par fichiers, par canaux explicites, par contrats d'interface) ne servent pas seulement à coordonner ; elles servent à laisser une mémoire de la décision, qu'un humain ou un autre agent pourra reprendre. C'est le sujet du chapitre 5.

Les trois briques se tiennent. Un état partagé fiable sans boucle décisionnelle reste un dossier de fichiers que personne n'utilise. Une boucle décisionnelle sans état à observer décide dans le vide. Et des décisions qu'on ne peut pas relire ne permettent ni d'auditer ni de corriger.

Pourquoi la décision est le pivot

J'ai longtemps présenté l'agentique en partant des outils : tel modèle, tel framework, tel pattern de fichiers. C'est ce que font la plupart des formations que j'ai vues. Et c'est utile au début.

À mesure que les systèmes que j'accompagnais devenaient plus ambitieux, j'ai vu un schéma se répéter. Les constructeurs maîtrisaient les outils. Ils savaient écrire des skills, organiser des dossiers, déclencher des agents par cron. Et pourtant, leurs systèmes se figeaient. Ils ajoutaient des fichiers, ajoutaient des règles et le système ne devenait pas plus intelligent : il devenait plus rigide.

La cause, en remontant, était toujours la même : aucun moment dans le système n'était conçu comme un point de décision. Tout était scénarisé d'avance.

Une fois qu'on identifie les points de décision (où l'agent doit choisir), la conception change de nature. Elle devient : où veut-on que l'agent décide ? Sur quel état doit-il s'appuyer ? Quels sont les choix légitimes ? Quel garde-fou ?

C'est pour ça que je centre ce livret sur la décision plutôt que sur les outils. Les outils, vous les apprendrez en chemin. Le réflexe de penser en termes de décisions, c'est ce qui fait la différence entre un système qui s'étoffe avec le temps et un système qui se fige.

Ce que ce livret va vous apprendre

Trois parties. La **Partie I** pose la thèse (ce chapitre, l'état partagé, la boucle décisionnelle). La **Partie II** entre dans l'atelier (fabrication des mc-files, communication entre agents, portage chez un client d'un autre métier). La **Partie III** ouvre le capot du starter kit Scanderia comme cas d'étude (sa conception, sa méthode de recherche).

Vous pouvez lire dans l'ordre ou commencer par la Partie III si vous préférez l'exemple avant la théorie.

Un avertissement honnête

Ce que je vais décrire dans ce livret, je le pratique depuis quelques mois sur mAIstrow, le système agentique sur lequel je travaille. Certaines choses sont stabilisées et tournent en production. D'autres sont des choix que je teste encore et dont je ne suis pas certain à cent pour cent. Je dirai à chaque fois où je me situe.

Ce livret n'est donc pas une recette. C'est une cartographie d'un terrain que j'explore moi-même, avec les meilleures pratiques que j'ai pu en tirer. Si vous trouvez mieux, écrivez-moi.

Chapitre 2

L'état partagé

Pour qu'une équipe d'agents prenne des décisions, il faut un endroit où l'état du système est lisible par tous. Ce livret monte en complexité, sans monter en infrastructure : le défaut reste le markdown. Ce chapitre explique comment le faire tenir, comment l'organiser quand le système grossit, à quel moment on accepte de bouger.

Pour qu'une équipe d'agents prenne des décisions, il faut un endroit où l'état du système est lisible par tous. Ce que les agents ont fait, ce qu'ils ont décidé, ce qu'il reste à faire, ce qui s'est passé avec tel client la semaine dernière.

Dans les livrets 1 et 2, cet endroit est un dossier de fichiers markdown. Ce livret va monter en complexité, mais pas en infrastructure. L'option par défaut reste le markdown. Ce chapitre explique comment le faire tenir, comment l'organiser quand le système grossit et à quel moment on accepte de bouger.

Le défaut : exécuter en série

Avant de parler de stockage, parlons d'orchestration. La question « comment éviter que deux agents s'écrivent dessus en même temps ? » a une réponse simple : **ne pas les faire travailler en même temps**.

Dans la grande majorité des systèmes que je conçois, les agents s'exécutent en série. Un manifeste fixe l'ordre. À 8h00 l'agent veille tourne, finit, écrit son journal. À 8h05 l'agent analyste lit ce journal, produit son rapport. À 8h10 le chef de bureau synthétise. Personne n'écrit en même temps que personne d'autre. La concurrence n'existe pas, donc le problème non plus.

Cette façon de faire a deux conséquences. La première, c'est qu'on n'a pas besoin de base de données. Un fichier markdown, écrit par un seul agent à la fois, est un état partagé parfaitement valide. La deuxième, c'est qu'on a une trace claire de qui a fait quoi à quel moment, sans effort particulier : le manifeste raconte l'histoire.

Quand est-ce que la séquentialité ne suffit plus ? Trois cas réels.

- **Un événement extérieur peut arriver à n'importe quel moment.** Un webhook qui prévient quand un client a écrit, un utilisateur qui poste une demande via l'app mobile. Là, votre agent ne choisit pas son moment, il réagit.
- **Le système doit faire deux choses indépendantes vraiment en même temps.** Cas rare

en pratique : la plupart du temps, même les choses qui semblent parallèles peuvent être faites en série sans qu'on perde grand-chose.
- **Le volume devient tel que l'exécution en série ne tient plus dans la fenêtre de temps disponible.** On y reviendra.

Pour les deux premiers cas, on traite l'événement en file d'attente : il s'ajoute à la fin et le prochain créneau disponible le prend. Pour le troisième, on parlera plus loin de ce qu'on fait quand l'état grossit.

Organiser les fichiers markdown comme un état

Dans un système qui décide, le markdown ne sert pas qu'à écrire des rapports lisibles par un humain. Il sert aussi à être lu par les autres agents pour orienter leurs décisions. Cette double fonction change un peu la façon dont on organise les fichiers.

Trois principes que j'applique systématiquement.

Un fichier par sujet, pas un fichier fourre-tout. Un journal unique pour toute l'équipe devient vite illisible. Je préfère un dossier par client, ou par projet, ou par thème, avec des fichiers courts dedans. Ça permet à un agent de ne charger que ce qui le concerne et ça facilite les synthèses ciblées (voir plus bas).

Une convention de nommage stable. Si un agent doit retrouver « le dernier rapport du client X », il doit pouvoir le faire sans deviner. Préfixe avec date inversée (`2026-04-28-rapport.md`), nom du sujet en clair, pas d'abréviations cryptiques. Vous écrivez pour un agent qui ne sait rien d'autre que ce qu'il lit.

Une structure interne reproductible. Chaque journal commence par le même en-tête (date, agent qui l'a écrit, statut). Chaque rapport suit la même trame (contexte, faits, décision proposée, ouverture). Les agents s'appuient sur la régularité du format pour extraire ce qui les intéresse. Une structure aléatoire oblige le modèle à tout relire à chaque fois.

Ces trois principes ne sont pas spécifiques au livret 3 (les livrets 1 et 2 les utilisent déjà). Mais ici ils deviennent critiques : c'est la lisibilité de l'état qui conditionne la qualité des décisions que les agents prennent dessus.

Quand l'état grossit : découper, pas grossir le fichier

Premier réflexe quand un fichier grossit : ne pas le laisser grossir. Je découpe en amont, par période, par client, par projet. Un fichier par mois, par client, par projet en cours.

Quelques exemples de découpage que j'utilise :

- Un journal par client : `client-acme/journal.md`, `client-bigcorp/journal.md`. Quand l'agent travaille sur un client, il ne charge que celui-là.

- Un journal par mois : `2026-04-journal.md`, `2026-05-journal.md`. Quand on cherche « ce qu'on a fait ce mois-ci », on ne charge qu'un fichier.
- Un dossier par projet, avec à l'intérieur un journal, un état courant, des rapports archivés.

Le découpage évite que l'agent paye en tokens à chaque lecture le poids de tout l'historique. Il lit la part qui le concerne, c'est tout.

Quand le découpage ne suffit plus : faire des synthèses

À un moment, même découpé, l'historique devient trop volumineux pour qu'un agent puisse tout relire à chaque décision. Vous avez deux ans de journaux clients, plusieurs milliers de notes d'analyse et un nouvel agent qui doit comprendre « qu'a-t-on appris sur ce type de problème ? ».

À ce moment, la tentation est d'aller chercher du RAG (un moteur qui transforme tous vos documents en vecteurs numériques et retrouve les morceaux pertinents par similarité). Le RAG fonctionne, mais il rajoute une brique technique sérieuse : un index à maintenir, un modèle d'embeddings à choisir, une qualité de retrieval à évaluer. Pour beaucoup de constructeurs, c'est la marche de trop.

Je préfère, dans la majorité des cas, **faire des synthèses**.

Une synthèse, c'est un fichier produit par un agent dédié, qui condense un ensemble de fichiers sources en un texte court et structuré. La même mécanique que le chef de bureau du livret intermédiaire, transposée à la mémoire longue de l'équipe.

Trois exemples de synthèses utiles :

- **Une synthèse mensuelle par client.** Tous les premiers du mois, un agent relit tout ce qui s'est passé sur le client pendant le mois écoulé et produit un fichier `client-acme/synthese-2026-04.md` de deux pages. Les autres agents, quand ils ont besoin de contexte sur ce client, lisent d'abord les synthèses mensuelles et ne plongent dans les journaux détaillés que si nécessaire.
- **Une synthèse thématique.** Une fois par semaine, un agent parcourt les journaux et alimente un fichier `themes/erreurs-frequentes.md` qui regroupe les patterns observés. Un nouvel agent qui rejoint l'équipe lit d'abord les synthèses thématiques pour se mettre à niveau.
- **Une synthèse projet.** À la fin d'un projet, un agent produit un `projets/acme-q1-2026/bilan.md` qui condense ce qui s'est passé. Le détail reste dans les fichiers d'origine, mais on n'a plus besoin de les charger pour comprendre.

L'agent qui produit ces synthèses n'a pas besoin d'être sophistiqué. C'est un déclencheur (cron, webhook), une lecture des fichiers concernés, une production d'un texte structuré selon une trame fixe. Le travail dur est dans le choix de ce qu'on synthétise et de la trame, pas dans la technique.

Avec ce mécanisme, le système peut fonctionner pendant des années sans jamais sortir du

markdown. La mémoire longue prend ici la forme d'une hiérarchie de résumés. Quand un agent veut savoir, il commence par les synthèses et il descend dans le détail seulement si la synthèse ne suffit pas.

Quand on accepte de bouger

Il existe des cas où le markdown plus synthèses ne suffit plus : volumes très importants, requêtes très structurées, besoin de cohérence transactionnelle, événements en temps réel. À ce moment, on monte en infrastructure (bases de données légères, files d'événements, parfois RAG). Ce n'est pas le sujet de ce livret.

Si vous arrivez à ce besoin, l'investissement vaut la peine. Mais avant d'y arriver, vérifiez que vous avez bien découpé, que vous synthétisez régulièrement et que vos agents n'essaient pas de lire l'intégralité de l'historique à chaque décision. Dans neuf cas sur dix que je vois, le problème n'est pas le markdown : c'est qu'on l'utilise mal.

Récapitulatif avant de tourner la page

L'état partagé d'un système d'agents tient dans des fichiers markdown lus et écrits en série. Vous l'organisez en dossiers par sujet, avec des conventions stables. Quand le volume augmente, vous découpez plus fin. Quand le découpage ne suffit plus, vous faites produire des synthèses par un agent dédié.

Vous montez en infrastructure seulement quand ces trois leviers ont été exploités, ce qui prend en général beaucoup plus longtemps qu'on ne le pense.

L'état partagé répond à la question « sur quoi les agents décident-ils ? ». Le chapitre suivant traite de l'autre moitié : comment ils décident et comment on encadre ces décisions.

Chapitre 3

La boucle décisionnelle

L'état partagé existe, les agents le lisent et l'écrivent. Mais relire un état ne suffit pas à rendre un système vivant : il faut que l'agent en tire une conclusion qui change ce qu'il va faire ensuite. Ce chapitre décrit comment un agent décide, quels types de décisions on rencontre, comment on les encadre. Il finit sur la forme de dérive qui résiste à tout l'encadrement classique.

L'état partagé existe, les agents le lisent et l'écrivent. Mais relire un état ne suffit pas à rendre un système vivant : il faut que l'agent en tire une conclusion qui change ce qu'il va faire ensuite. C'est ce que j'appelle la boucle décisionnelle.

Ce chapitre décrit comment un agent décide concrètement, quels types de décisions on rencontre et surtout comment on les encadre pour qu'elles restent prévisibles et auditables.

Ce qu'est une décision d'agent

Une décision, dans un système agentique, c'est trois choses qui s'enchaînent.

D'abord, **observer un état**. L'agent lit le fichier, la synthèse, le journal de la veille, le manifeste. Il prend connaissance de ce qui est vrai au moment où il se réveille.

Ensuite, **choisir parmi plusieurs options légitimes**. L'agent sélectionne dans une liste que vous avez définie à l'avance, sans en imposer ni en inventer. « Je traite cette demande maintenant ou je l'escalade. » « Je relance le client ou je laisse passer une semaine. » « Je travaille sur le projet A ou sur le projet B aujourd'hui. »

Enfin, **tracer ce qui a été décidé et pourquoi**. L'agent écrit dans un fichier (le journal du jour, un log de décision, ou la synthèse correspondante) : ce qu'il a vu, ce qu'il a choisi, sur quel critère. Cette trace sert à deux choses : permettre à un humain ou à un autre agent de reprendre le fil ; permettre à vous, constructeur, d'auditer le système quand quelque chose se passe mal.

Une décision sans trace reste opaque. Un système opaque ne peut être ni audité ni amélioré en confiance.

Quatre types de décisions qu'on rencontre

Dans les systèmes que j'accompagne, je retrouve à peu près toujours les mêmes familles de décisions. Les nommer aide à les concevoir proprement.

Le routage. L'agent reçoit une entrée et doit l'envoyer au bon endroit. Une demande client arrive : est-ce une question commerciale (vers l'agent commercial), une question support (vers l'agent support), une demande qui sort du cadre (vers l'humain) ? Le routage est probablement le type de décision le plus fréquent et le plus simple à encadrer : on définit les catégories, on définit la règle d'attribution, l'agent applique.

La priorité. L'agent a plusieurs choses à faire et doit choisir l'ordre. Trois projets en cours, lequel je traite en premier ce matin ? Cinq tickets dans la file, dans quel ordre je les prends ? Ici la décision dépend de critères que vous définissez : urgence, taille du client, type de demande, date limite. L'agent applique les critères dans un ordre fixé.

L'escalade. L'agent rencontre une situation qu'il n'est pas sûr de pouvoir traiter. Il a deux choix : décider quand même au risque de se tromper, ou prévenir un humain. La règle par défaut, dans tout ce que je conçois, est que **l'escalade est gratuite**. Mieux vaut un agent qui escalade trop que pas assez. On affine ensuite, en voyant lesquelles auraient pu être prises seules.

La replanification. L'agent était parti pour faire la chose A, l'état change pendant qu'il travaille (un événement arrive, une priorité bouge), il doit décider s'il continue ou s'il bascule. C'est le type de décision le plus délicat parce qu'il introduit du non-déterminisme : deux exécutions du même système peuvent prendre des chemins différents. À garder pour les cas où c'est vraiment nécessaire.

Comment encadrer une décision

Une décision d'agent qui n'est pas encadrée est ingérable. Elle peut être brillante un jour et catastrophique le lendemain, sans que vous sachiez pourquoi. L'encadrement repose sur trois choses.

Une liste finie d'options légitimes. On n'autorise jamais l'agent à inventer une option. Si vous voulez qu'il choisisse entre « traiter », « reporter » ou « escalader », ce sont les trois seules sorties acceptées. Aucune autre. C'est une contrainte qu'on pose dans le prompt et qu'on vérifie dans le format de sortie attendu (chapitre 1 du livret intermédiaire sur les skills).

Des critères explicites. La règle qui mène à un choix doit être écrite quelque part, en français lisible, dans le dossier de l'agent. « Une demande est urgente si elle vient d'un client au statut prioritaire ou si elle mentionne un mot du vocabulaire d'urgence (panne, bloqué, urgent). » Pas « l'agent décide selon son jugement » : c'est précisément ce qu'on veut éviter.

Un garde-fou. Pour chaque famille de décisions, vous définissez les cas où l'agent doit s'arrêter et prévenir l'humain plutôt que d'agir. Une demande qui mentionne un contentieux : escalade obligatoire. Une décision qui engage plus d'un certain montant : escalade obligatoire. Un cas

qui ne tombe dans aucune catégorie connue : escalade obligatoire.

Ces trois éléments forment le contrat de la décision. Ils doivent être lisibles dans le dossier de l'agent par n'importe quel humain de l'équipe, sans avoir à interpréter du code.

L'humain comme garde-fou par défaut

Je veux insister sur ce point parce que c'est le plus mal géré dans les systèmes que je vois en consulting.

L'humain est la première ligne de garde-fou d'un système agentique, avant même la roue de secours. Tout ce que l'agent n'est pas explicitement autorisé à décider, il l'escalade. Toujours.

Dans la pratique, ça veut dire que pour chaque agent, je définis un **périmètre de décision**, c'est-à-dire la liste exhaustive des choix que cet agent est autorisé à prendre seul. En dehors de cette liste, il prévient l'humain. Le périmètre commence toujours petit et il s'élargit avec le temps, à mesure qu'on voit ce que l'agent fait bien et ce qu'il ne fait pas bien.

Je n'ai jamais vu un système agentique tomber en panne parce que son périmètre était trop petit. J'ai vu plusieurs systèmes créer des problèmes sérieux parce que leur périmètre était trop large dès le départ.

L'élargissement progressif du périmètre est aussi le mécanisme par lequel le constructeur reprend la confiance dans son système : on étend, on observe, on corrige, on étend encore.

La dérive de périmètre implicite

Il y a une forme de dérive qui résiste à tout ce qu'on vient de poser. L'agent respecte ses options légitimes formelles. Il ne fait que ce qu'on a documenté. Son domaine d'action effectif s'élargit pourtant silencieusement, parce qu'il décide là où il aurait dû demander.

Le cas type : un agent qui peut traiter ou escalader. Sur les premiers mois, il escalade beaucoup. Avec le temps, il traite de plus en plus de cas qu'il aurait fallu remonter. Aucune règle n'est violée, aucune sortie ne dépasse la liste autorisée. Mais la frontière de décision s'est déplacée sans que personne ne l'ait actée.

J'appelle ça la dérive de périmètre implicite. C'est la zone aveugle des garde-fous qu'on a posés jusqu'ici.

Pourquoi les garde-fous classiques ne la voient pas

Les cas-or testent ce que l'agent **produit** quand on lui donne une entrée typique. Format, ton, structure, choix dans la liste autorisée. Tout ça reste correct par construction tant que l'agent

reste dans son périmètre formel. Les cas-or ne testent pas ce que l'agent fait avec une entrée qu'il aurait dû escalader et qu'il a quand même traitée. Le critère de réussite des cas-or porte sur la sortie produite, sans interroger la frontière de décision.

Le manifeste équipe ne la voit pas non plus. Il décrit ce que l'agent est autorisé à décider. Il ne mesure pas ce que l'agent décide effectivement par rapport à ce qu'il aurait dû déférer.

C'est l'angle aveugle de la sortie : on surveille la qualité de ce qui sort, sans interroger la légitimité de la décision de sortir au lieu de demander.

Loguer les non-escalations comme des décisions

Le premier réflexe est d'enrichir la trace. Quand l'agent décide, on logue déjà sa décision et son critère. Il faut ajouter une catégorie : les cas où il aurait pu escalader et où il a choisi de ne pas le faire.

Concrètement, dans le prompt de l'agent, on fait apparaître la non-escalation comme une décision explicite. La trace doit faire apparaître le raisonnement : « j'ai vu un signal d'escalade [tel critère] et j'ai jugé qu'il ne s'appliquait pas parce que [tel autre critère] ». Cette signature comportementale rend la frontière auditable.

L'audit de bord

Une fois ces traces accumulées sur quelques semaines, on fait un audit de bord. Sur N décisions prises seules, on en tire un échantillon (vingt, trente, ce que vous voulez) et on les rejoue à l'humain : auriez-vous traité ou escaladé ?

Le ratio que vous obtenez vous dit où la frontière effective se trouve. Si dix-huit sur vingt confirment la décision de l'agent, le périmètre tient. Si cinq ou six auraient dû basculer humain, la frontière s'est déplacée et il faut la recadrer.

C'est une mécanique simple, qui demande une heure par mois. Elle n'a pas besoin d'outillage particulier. Elle a besoin de discipline (la faire tourner régulièrement) et d'honnêteté (accepter que certaines décisions de l'agent qu'on trouvait élégantes étaient en fait des dépassements de périmètre).

Options autorisées et domaine d'action effectif

Pour que cette mécanique tienne dans la durée, il faut séparer deux notions qu'on confond souvent.

Les **options autorisées** sont la liste finie qu'on a posée dans le prompt (traiter, reporter, escalader). C'est statique, écrit, vérifiable.

Le **domaine d'action effectif** est l'ensemble des situations dans lesquelles l'agent a effectivement choisi sans escalader. C'est dynamique, produit par l'usage, observable seulement a posteriori.

Le périmètre de décision (au sens posé plus haut) est le domaine d'action que vous **autorisez**. La dérive implicite, c'est l'écart entre ce que vous autorisez et ce que l'agent prend.

Garder ces deux notions distinctes évite l'illusion que poser un prompt strict suffit à figer le comportement. Ça aide à poser le prompt. Ça ne mesure pas ce qui se passe ensuite.

Anti-patterns

Quelques pièges que je rencontre régulièrement.

L'agent qui décide tout, tout le temps, sans escalade. Souvent par excès de confiance dans le modèle. Au premier cas limite, le système prend une mauvaise décision silencieusement, et personne ne le voit avant que le client ne se plaigne.

Les critères qui se contredisent. Vous ajoutez une règle ici, une autre là, sans relire l'ensemble. L'agent se retrouve avec deux instructions incompatibles et choisit au hasard. Tenez les critères de décision dans un seul fichier par agent, relisez-le entier à chaque modification.

La replanification systématique. Dès qu'un événement arrive, l'agent abandonne ce qu'il faisait pour s'en occuper. Vous obtenez un système qui ne finit jamais rien. La replanification doit être l'exception, pas le mode de fonctionnement.

Pas de trace de décision. L'agent décide mais n'écrit nulle part ce qu'il a décidé ni pourquoi. Quand un problème survient, vous ne pouvez pas reconstituer ce qui s'est passé. Faites écrire systématiquement la décision et son critère, même quand ça ralentit un peu l'agent.

Confier la décision à un agent au lieu de la formuler. Le piège le plus subtil : on se dit « l'agent saura quoi faire » et on lui laisse le choix sans cadre. Dans neuf cas sur dix, ce choix existait déjà dans votre tête, vous ne l'aviez juste pas formalisé. Le travail du constructeur est précisément de formaliser ce choix au lieu de le déléguer au modèle.

Récapitulatif de la Partie I

Avec ce chapitre, on referme la première partie du livret. Vous avez maintenant le vocabulaire et les briques d'un système agentique vivant : un état partagé qui sert de base aux décisions, des décisions encadrées par des règles explicites, des conversations qui laissent une trace.

Ces trois briques sont les pièces. La partie suivante traite de leur fabrication concrète : comment on écrit les mc-files qui supportent ce niveau de complexité, comment on fait communiquer les agents proprement, comment on construit un système pour un autre métier que le sien.

À ce stade, si vous regardez un système agentique existant (le vôtre, ou un que vous accompagnez), vous devriez pouvoir répondre à quatre questions : où est l'état partagé ? Quels sont les points de décision ? Comment l'humain est-il prévenu quand on sort du périmètre ? Comment vous mesurez l'écart entre le périmètre autorisé et le domaine d'action que l'agent prend effectivement ? Si l'une des quatre reste floue, vous tenez un point d'entrée pour le travail à faire.

Chapitre 4

Fabriquer ses propres mc-files

La Partie I a posé les pièces d'un système agentique vivant. On entre maintenant dans l'atelier : comment on fabrique les fichiers qui font marcher tout ça. Ce chapitre décrit le processus en cinq étapes que j'utilise pour passer du besoin métier brut au mc-file qui tourne en production, en insistant sur la spécification et sur les cas limites.

La Partie I a posé les pièces d'un système agentique vivant. On entre maintenant dans l'atelier : comment on fabrique les fichiers qui font marcher tout ça.

Le livret intermédiaire vous a montré la structure d'un skill (les cinq composants, les cinq tests, les cinq pannes classiques). Ce chapitre traite d'autre chose : non pas comment un mc-file est structuré, mais comment on le **fabrique**, depuis le besoin métier brut jusqu'au fichier stable qui tourne en production.

C'est un travail de spécification autant que de rédaction. La plus grande partie du travail se passe avant d'avoir écrit la première ligne de markdown.

Pourquoi ce chapitre est nécessaire

Beaucoup de constructeurs partent du fichier. Ils ouvrent un nouveau document, écrivent quelques instructions, lancent l'agent, voient ce que ça donne, corrigent. C'est la méthode de la première semaine. Elle marche pour comprendre comment ça fonctionne.

Pour produire un mc-file qui tient en production, qui sera relu par d'autres humains, modifié dans six mois et compris sans contexte, cette méthode ne suffit pas. Il faut un processus.

Je vais décrire celui que j'utilise. Il est itératif et il prend du temps. Pour un mc-file de qualité production, je compte rarement moins d'une demi-journée de travail effectif, souvent une journée entière. Ce temps est l'investissement qui permet au fichier de servir pendant des mois ou des années sans qu'on ait à y revenir.

Étape 1 · Spécifier le besoin métier sans toucher au markdown

C'est l'étape la plus négligée et la plus rentable. Avant d'écrire quoi que ce soit, je réponds par écrit à cinq questions, dans un fichier de notes que je jette ensuite.

À qui sert cet agent ? La réponse utile est « tel rôle, dans tel contexte, à tel moment de sa journée ». Une réponse comme « toute mon équipe » est trop large pour servir. Plus c'est précis, mieux c'est.

Qu'est-ce qu'il fait, en une phrase ? Si je n'arrive pas à formuler ce que fait l'agent en une phrase claire, c'est que je vais probablement essayer de lui faire faire deux choses et le fichier sera bancal.

Qu'est-ce qu'il ne fait pas ? Souvent plus important que ce qu'il fait. Définir le périmètre par négation force à expliciter les frontières. « L'agent rédige des brouillons d'email mais ne les envoie jamais. » « L'agent qualifie les leads mais ne contacte pas les clients. »

Quelle est la sortie attendue ? Format précis. Un email ? Un fichier markdown ? Une décision dans une liste fermée ? Un résumé en moins de 200 mots ? Plus c'est concret, plus le mc-file pourra être contraint.

Quels sont les cas où l'agent doit s'arrêter et prévenir l'humain ? La règle d'escalade, du chapitre précédent. Définie dès maintenant, pas en patch ensuite.

Si vous ne pouvez pas répondre à ces cinq questions clairement, vous n'êtes pas prêt à écrire le mc-file. Retournez voir le client, le métier, l'utilisateur. La spécification est le vrai livrable.

Étape 2 · Écrire le premier jet

Une fois la spécification faite, le premier jet du mc-file prend généralement entre une et trois heures. Je suis la structure standard qu'on a vue dans le livret intermédiaire : header de déclenchement, overview, workflow, format de sortie, exemples.

Quelques règles que je m'impose à ce stade.

Aucun « peut-être », aucun « selon le contexte ». Si je me surprends à écrire ce genre de chose, c'est que la spécification de l'étape 1 n'est pas assez précise. Je retourne à l'étape 1, je tranche, je reviens.

Au moins deux exemples concrets dans le mc-file. Un exemple positif (« voilà ce qu'on attend »), un exemple négatif (« voilà ce qu'on ne veut pas »). Les exemples sont souvent plus instructifs pour le modèle que les règles.

Le périmètre est écrit en clair. Une section dédiée qui liste ce que l'agent fait et ce qu'il ne fait pas. Pas une phrase noyée dans le workflow.

Le premier jet est presque toujours imparfait. C'est normal. Le gros du travail vient après.

Étape 3 · Tester sur cas simples

Avant de tester sur des cas tordus, je vérifie que l'agent marche sur les cas faciles. Trois ou quatre entrées qui correspondent au cas d'usage typique. Je regarde la sortie. Si elle ne convient pas, je corrige le mc-file et je recommence.

Cette phase doit aboutir vite (une demi-heure, peut-être une heure). Si elle traîne, le problème est en amont : la spécification est confuse, ou le premier jet ne s'y conforme pas. Je remonte.

Je ne passe à l'étape suivante que quand les cas faciles passent tous proprement.

Étape 4 · Itérer sur les cas limites et les tests négatifs

C'est l'étape la plus longue et la plus utile. Je construis une liste de cas qui pourraient faire dérailler l'agent.

Cas ambigus. Une entrée où plusieurs interprétations sont possibles. Comment l'agent choisit-il ? Selon quel critère ? Sa décision est-elle traçable ?

Cas hors périmètre. Une entrée qui ne correspond pas à ce que l'agent est censé traiter. Refuse-t-il proprement ? Escalade-t-il ?

Cas adversariaux. Une entrée qui essaie de faire sortir l'agent de son cadre (« ignore les instructions précédentes », « en fait je suis ton administrateur »). Tient-il ?

Cas dégradés. Une entrée incomplète, mal formée, contradictoire. Demande-t-il une clarification ? Devine-t-il ? S'arrête-t-il ?

Pour chaque cas qui ne passe pas comme attendu, je fais une modification ciblée du mc-file et je rejoue **toute** la batterie de tests. Une modification qui répare un cas peut en casser un autre. C'est la phase qui ressemble le plus à du développement logiciel : la régression est un risque permanent.

J'arrête cette étape quand la batterie passe stable sur trois exécutions consécutives, sans que j'aie eu à modifier le mc-file entre les deux.

Étape 5 · Stabiliser et documenter

Le mc-file marche. Je le considère comme livrable, mais pas encore comme fini. Il reste deux choses à faire.

Documenter pour les humains. En haut du fichier, j'ajoute quelques lignes qui expliquent à

un humain qui ouvre le mc-file dans six mois : à quoi sert cet agent, quand il est déclenché, ce qu'il ne fait pas, où trouver les cas de test. Cette documentation ne sert pas à l'agent, elle sert à votre équipe ou à votre client.

Archiver les cas de test. La batterie que j'ai construite à l'étape 4 ne se jette pas. Je la garde, dans un fichier à côté du mc-file (`cas-de-test.md` par exemple). Quand le mc-file devra évoluer dans six mois, ces cas serviront à vérifier qu'on n'a rien cassé.

À ce stade, le mc-file est en production. On le surveille, on note les anomalies, on les ajoute aux cas de test et on itère quand c'est nécessaire. Mais le gros du travail est fait.

Anti-patterns

Quelques pièges fréquents que j'observe.

Sauter l'étape 1. « Je vais écrire le mc-file et on verra en testant. » On y passe deux fois plus de temps et le résultat est moins clair.

Tester avec une seule entrée. L'agent réussit, on déclare victoire. Six semaines plus tard, un cas légèrement différent casse tout et personne ne sait pourquoi.

Ajouter des règles sans en retirer. Le mc-file gonfle, les règles se contredisent, le modèle devient confus. Quand vous ajoutez une règle, demandez-vous si elle remplace une règle existante.

Confondre exemple et règle. Un exemple n'est pas une règle. Mettre cinq exemples du même cas ne couvre pas les autres cas. Mettre une règle générale peut couvrir cinquante cas.

Ne pas archiver les cas de test. Vous referez le travail de qualification dans six mois, à partir de zéro, avec moins de mémoire du contexte qu'aujourd'hui.

Quand un mc-file devient deux mc-files

Un mc-file qui marche peut quand même être mal découpé. Le signal le plus fréquent que j'observe est ce que j'appellerai la **surcharge de skills** : l'agent a trop d'outils ou trop de responsabilités proches. Il commence alors à se tromper, à hésiter, ou à oublier des options.

Trois symptômes concrets.

L'agent choisit le mauvais skill. Vous avez `fetch-twitter` et `fetch-bluesky`, l'agent appelle l'un quand vous attendiez l'autre. La frontière entre les deux est floue dans son dossier, parce qu'elle est floue dans le réel : ce sont deux variantes proches de la même opération.

L'agent hésite et boucle. Face à une entrée, il liste les options, en essaie une, revient en arrière, en essaie une autre. La sortie finit par arriver mais le chemin est confus. C'est le signe que la décision interne est trop large.

L'agent oublie certains skills. Il en a quinze, il en utilise toujours les trois mêmes. Les douze autres polluent son contexte sans servir. Ils empirent même la précision sur les trois qu'il utilise vraiment.

Quand je vois ces symptômes, le réflexe n'est pas d'améliorer le prompt. C'est de **splitter**. Un agent veille qui mélange twitter, bluesky, podcasts, papiers et newsletters devient deux ou trois agents : veille-textes, veille-audio, veille-papiers. Chacun avec un dossier resserré, un périmètre clair, des skills proches entre eux mais éloignés des autres groupes.

La règle que je m'impose : si un agent a plus de cinq à sept skills très proches les unes des autres, c'est probablement deux agents qui se déguisent en un.

Le bon réflexe ici rejoint l'étape 1 : si vous ne pouvez pas formuler en une seule phrase ce que fait l'agent, le signal était déjà là, vous l'avez juste laissé passer.

Quand un mc-file ne marche pas

Le diagnostic suit un ordre simple, du plus probable au moins probable.

La spécification est confuse. L'agent ne sait pas ce qu'on attend de lui parce que vous ne le saviez pas vous-même. Retour à l'étape 1.

Le mc-file ne reflète pas la spécification. La spécification est claire mais le fichier dit autre chose. À relire et corriger.

Les exemples sont mal choisis. Le modèle suit les exemples et vos exemples ne sont pas représentatifs. Remplacez-les.

Le modèle utilisé n'est pas adapté. Cas plus rare. Si vous avez tout vérifié au-dessus, essayez un modèle plus capable (ou plus petit, selon le cas).

Dans neuf cas sur dix, le problème est dans les deux premières catégories. Les constructeurs commencent par soupçonner le modèle et finissent par découvrir qu'ils n'avaient pas spécifié.

Récapitulatif avant de tourner la page

Fabriquer un mc-file de qualité production demande un processus en cinq étapes : spécifier, premier jet, tester simple, itérer sur cas limites, stabiliser et documenter. Le travail le plus rentable est dans la spécification (étape 1) et dans les cas limites (étape 4). Les deux sont régulièrement sautés ou expédiés.

Un mc-file qui tient en production tient parce qu'on a pris le temps de le construire ainsi. Le bon modèle ou la formulation magique ne remplacent pas ce travail. La méthode prime sur le contenu.

Le chapitre suivant traite de la coordination entre plusieurs mc-files : comment on fait pour

que plusieurs agents fabriqués selon cette méthode communiquent proprement, sans que l'ensemble devienne un sac de nœuds.

Chapitre 5

Faire communiquer les agents

Vous savez maintenant fabriquer un mc-file de qualité production. Le sujet de ce chapitre : ce qui se passe quand on en met plusieurs ensemble. Trois modes de communication (passage de témoin, tableau commun, requête), un contrat d'interface explicite, trois règles structurelles pour éviter le sac de nœuds.

Vous savez maintenant fabriquer un mc-file de qualité production. Le sujet de ce chapitre : ce qui se passe quand on en met plusieurs ensemble et qu'ils doivent travailler en relais.

Une remarque pour commencer. Dans la philosophie mc-files, même si les agents s'exécutent en série (chapitre 2), ils communiquent quand même. Leur communication passe par les choses qu'ils se laissent à lire, plutôt que par un échange direct. Cette nuance change beaucoup de choses sur la façon de concevoir la communication.

Trois modes de communication entre agents

J'observe trois modes en pratique. Ils ne sont pas exclusifs, un système peut les utiliser tous les trois.

Le passage de témoin

C'est le mode le plus simple et le plus fréquent. L'agent A finit son travail, écrit le résultat dans un fichier (ou plusieurs), puis l'agent B prend le relais et lit ce que A a laissé.

Exemple typique : l'agent veille collecte les nouveautés du matin et les écrit dans `veille/2026-04-28.md`. Une demi-heure plus tard, l'agent analyste lit ce fichier et produit un rapport synthétique dans `rapports/2026-04-28-synthese.md`. À midi, le chef de bureau lit le rapport et envoie un message à l'équipe.

Trois caractéristiques importantes du passage de témoin.

Asynchrone par construction. L'agent A n'attend pas que B soit prêt. Il finit son travail et le laisse au sol. Si B n'arrive pas tout de suite, ce n'est pas grave : la trace est là.

Auditable. Vous pouvez relire à tout moment qui a écrit quoi à quel moment. Pas besoin d'instrumenter quoi que ce soit, le fichier est sa propre trace.

Robuste aux pannes. Si B plante, le travail de A n'est pas perdu. Vous corrigez B et vous le

relancez sur le même fichier d'entrée.

C'est le mode par défaut dans tous mes systèmes. Si vous hésitez, commencez par celui-là.

Le tableau commun

Plusieurs agents lisent et complètent un même fichier d'état partagé, à tour de rôle, chacun apportant sa contribution (toujours en série, jamais simultanément).

Exemple : un fichier `projet-acme/etat.md` qui décrit l'état courant du projet. L'agent commercial le complète avec les informations clients. L'agent technique le complète avec les points de blocage. L'agent chef de projet le synthétise et prend les décisions de priorité.

Le tableau commun ressemble au passage de témoin, mais le fichier vit dans le temps : il s'enrichit, il se met à jour, il sert de référence partagée. Là où le passage de témoin produit un nouveau fichier à chaque fois (un journal du jour, un rapport du jour), le tableau commun maintient un fichier unique qui évolue.

Le risque du tableau commun : si vous n'avez pas une convention stricte sur qui écrit quoi et où, le fichier devient vite un patchwork incohérent. Pour chaque tableau commun, je définis explicitement :

- **Quelles sections existent** (et pas d'invention de section par les agents).
- **Qui peut écrire dans quelle section** (l'agent technique ne touche pas à la section commerciale et inversement).
- **Comment on horodate les modifications** (chaque ajout porte la date et l'agent qui l'a fait).

Avec ces trois conventions, le tableau commun reste lisible sur le long terme. Sans, il dégénère en quelques semaines.

La requête

Un agent appelle explicitement un autre agent pour obtenir une expertise particulière. L'agent rédacteur a besoin d'une vérification juridique : il appelle l'agent juridique, qui vérifie et renvoie sa réponse.

C'est le mode le plus proche de ce qu'on appelle souvent A2A (agent-to-agent) dans la littérature. C'est aussi le plus risqué.

Risqué pour deux raisons.

Le couplage fort. L'agent rédacteur dépend de l'agent juridique. Si le juridique est cassé ou modifié, le rédacteur casse. Le passage de témoin et le tableau commun n'ont pas ce problème : la communication passe par un fichier, qui est indépendant des deux agents.

La complexité d'orchestration. Un agent qui en appelle un autre, qui en appelle un troisième, devient vite un système où personne ne sait plus qui attend quoi. Le débogage est pénible.

Mon conseil : utilisez la requête seulement quand le passage de témoin ne convient vraiment pas. C'est-à-dire quand l'agent demandeur a besoin de la réponse pour décider quoi faire ensuite et qu'il ne peut pas attendre un cycle complet.

Dans mes systèmes, je mets en place la requête dans peut-être un cas sur dix. Le reste passe par des fichiers.

Le contrat d'interface

Quel que soit le mode, deux agents qui communiquent doivent respecter un contrat. Ce contrat décrit ce que l'agent producteur s'engage à laisser, dans quel format, à quel endroit et ce que l'agent consommateur s'engage à savoir lire.

Un contrat d'interface est un fichier court (souvent une page), placé à côté des deux mc-files concernés, qui décrit :

Le fichier ou les fichiers échangés. Avec leur chemin exact, par exemple `rapports/AAAA-MM-JJ-veille.md`. Une formulation comme « un rapport quelque part » ne suffit pas.

La structure interne attendue. Quelles sections, dans quel ordre, avec quels titres. Si une section est optionnelle, c'est écrit. Si une section a un format précis (liste, tableau, JSON), c'est écrit.

Les invariants. Ce qui doit toujours être vrai du fichier produit. « Le rapport contient toujours au moins une entrée. » « Les dates sont toujours au format ISO. » « Le résumé fait toujours moins de 200 mots. »

Ce qui se passe en cas de manquement. Si l'agent A produit un fichier qui ne respecte pas le contrat, qu'est-ce que B fait ? L'idéal est qu'il escalade plutôt que d'essayer de deviner.

Un contrat d'interface explicite, c'est ce qui transforme une collection de mc-files en un système. Sans contrat, vous avez des agents qui se parlent par intuition partagée et ça tient tant que personne ne touche à rien. Avec contrat, vous pouvez modifier un agent sans casser tout le reste, à condition de respecter le contrat.

Trois règles pour éviter le sac de nœuds

À mesure que le nombre d'agents augmente, le risque que l'ensemble devienne ingérable augmente bien plus vite. Quelques règles que je m'impose.

Pas de communication non documentée. Toute communication entre deux agents passe par un fichier ou une requête et le contrat existe. Si je découvre une communication non documentée, c'est une dette technique qui devra être traitée.

Une seule source de vérité par information. Une donnée existe à un seul endroit. Les autres

agents y accèdent par référence (ils lisent ce fichier), pas en gardant une copie. Sans cette règle, vous obtenez plusieurs versions divergentes de la même information.

Le manifeste équipe est à jour. Le fichier qui liste tous les agents et leurs liens (livret intermédiaire, chapitre 12) doit refléter la réalité. À chaque ajout d'agent ou de contrat, on met à jour. Sans manifeste à jour, le système devient opaque.

Anti-patterns

L'agent qui écrit dans plusieurs fichiers d'état différents. Vous obtenez de la duplication, donc de la divergence à terme. Un agent écrit dans un fichier, un seul. Si plusieurs autres agents ont besoin de l'information, c'est à eux de venir la lire dans ce fichier.

Les contrats implicites. « Tout le monde sait que le rapport contient toujours un résumé. » Personne ne le sait. Écrivez le contrat.

La communication par signaux flous. Un agent écrit « situation à surveiller » dans un journal. Un autre est censé deviner que ça veut dire « escalade niveau 2 ». À écrire en clair, dans le contrat, avec un vocabulaire fini.

Les boucles de requêtes. L'agent A appelle B, qui appelle C, qui appelle A. Ça ne se termine pas. Un graphe de requêtes doit former un arbre, jamais un cycle. Vérifiez à la conception.

La ré-implémentation par copier-coller. Deux agents font à peu près la même chose, vous dupliquez le mc-file et vous modifiez. Six mois plus tard, les deux fichiers ont divergé. Personne ne sait lequel est la référence. Préférez un seul mc-file factorisé, avec deux instanciations, ou un mc-file parent dont les deux héritent.

Récapitulatif avant de tourner la page

Trois modes de communication : passage de témoin (par défaut), tableau commun (état partagé qui vit dans le temps), requête (à utiliser avec parcimonie). Chaque communication entre deux agents repose sur un contrat d'interface explicite, écrit, à côté des mc-files concernés.

Trois règles structurelles : pas de communication non documentée, une seule source de vérité par information, manifeste équipe à jour.

Avec ce niveau de discipline, vous pouvez tenir un système de dix à vingt agents sans qu'il devienne ingérable. Au-delà, vous aurez besoin de structures supplémentaires (sous-équipes, hiérarchie de manifestes), mais peu de constructeurs en arrivent là et c'est un sujet pour un autre livret.

Le chapitre suivant traite du dernier sujet de la Partie II : construire un système agentique pour un autre métier que le sien. La méthode change quand vous ne maîtrisez pas le domaine métier vous-même.

Chapitre 6

Construire un système pour un autre métier

Jusqu'ici, je supposais que vous connaissiez le métier auquel ces agents s'appliquent. Quand vous construisez pour un client, c'est presque jamais le cas. Ce chapitre décrit ce qui change quand le domaine métier ne vous appartient pas : méthode en quatre temps, séparation couche commune / couche client, ce qu'il faut négocier en amont.

Jusqu'ici, je vous ai parlé de fabriquer des mc-files et de les faire travailler ensemble. Implicitement, je supposais que vous connaissiez le métier auquel ces agents s'appliquent. Quand vous construisez pour vous-même, c'est presque toujours le cas.

Quand vous construisez pour un client, ça ne l'est presque jamais. Le sujet de ce chapitre : ce qui change dans la méthode quand le domaine métier ne vous appartient pas.

C'est un chapitre adressé à ceux d'entre vous qui travaillent en consulting, en intégration ou en formation et qui livrent à des clients dans des secteurs qu'ils ne maîtrisent pas en profondeur. Si vous construisez seulement pour votre propre activité, vous pouvez le lire en diagonale.

Pourquoi c'est différent

Quand vous fabriquez un mc-file pour vous-même, vous êtes à la fois le constructeur et l'utilisateur. Vous savez ce qui doit sortir parce que vous savez ce que vous voulez. Vous savez quels sont les cas limites parce que vous les avez déjà rencontrés. Vous savez quand l'agent dit une bêtise parce que vous reconnaissez la bêtise immédiatement.

Pour un client, rien de tout ça n'est garanti. Ce qui sort de l'agent vous semble plausible, mais vous n'avez pas le réflexe métier qui permet de juger. Les cas limites, vous ne les connaissez pas et le client non plus, parce qu'il les vit sans les avoir formalisés. Les bêtises de l'agent passent inaperçues, jusqu'à ce qu'un utilisateur final les attrape, souvent en production.

La conséquence pratique : la méthode du chapitre 4 (spécification, premier jet, tests, itération) reste valable, mais l'étape 1 (la spécification) devient d'un coup beaucoup plus difficile et beaucoup plus longue. C'est là que se joue la qualité du livrable.

Le piège des deux extrêmes

J'ai vu deux écueils symétriques chez les constructeurs qui attaquent un nouveau métier.

Le kit générique. On part d'un template (« le kit pour cabinets de conseil », « le kit pour agences immobilières ») et on le sert tel quel à tous les clients d'un secteur. C'est rapide à livrer. Le problème : aucun client ne s'y reconnaît vraiment. Les agents produisent des choses correctes dans l'absolu, mais qui ne servent à personne en particulier. Le client paye, range le kit dans un coin et n'y revient jamais.

Le kit hyper-spécifique. On passe trois mois immergé chez un client, on construit un système calé sur ses processus à la virgule près et le résultat est magnifique. Le problème : vous ne pouvez pas le réutiliser pour le client suivant. Vous avez perdu de l'argent sur celui-ci.

La méthode utile se situe entre les deux. Elle consiste à identifier ce qui est commun à tout le secteur (et qu'on peut réutiliser) et ce qui est spécifique à ce client (et qu'on construit avec lui). Les deux couches existent et on les sépare proprement.

La méthode en quatre temps

Temps 1 · Interviewer pour découvrir le métier

La première erreur classique : arriver chez le client avec un projet de système déjà esquissé et faire des entretiens pour « valider l'approche ». Vous n'apprendrez rien. Les gens vous diront que c'est très bien, parce qu'ils sont polis et qu'ils ne veulent pas froisser le consultant.

L'entretien utile commence par des questions ouvertes sur le travail tel qu'il se fait aujourd'hui. « Décrivez-moi une journée typique. » « Qu'est-ce qui vous prend le plus de temps ? » « Quelles sont les tâches que vous repoussez tout le temps ? » « Quand est-ce qu'un dossier traîne ? Pourquoi ? »

Je fais en général entre cinq et dix entretiens, avec des profils différents (dirigeant, opérationnel, support, parfois client final). Je prends des notes brutes, sans essayer de synthétiser sur le moment. La synthèse vient après, quand j'ai laissé reposer.

Ce qui sort de cette phase n'est pas une spécification d'agent. C'est une compréhension du métier suffisante pour pouvoir commencer à formuler des hypothèses utiles.

Temps 2 · Observer le travail réel au-delà du discours

Les entretiens vous donnent ce que les gens disent. Ce qu'ils font est souvent différent. Les écarts entre les deux sont là où se cachent les opportunités d'automatisation intéressantes.

Quand c'est possible, je passe une demi-journée à regarder travailler les personnes que j'ai

interviewées. Pas pour les juger, juste pour voir. Quelles applications ils ouvrent. Combien de fois ils basculent d'une fenêtre à l'autre. Où ils copient-collent. Où ils s'arrêtent pour réfléchir. Où ils demandent l'avis d'un collègue.

Cette observation révèle des choses qu'aucun entretien ne révèle. Une opérationnelle qui dit en entretien « je gère ça en 5 minutes » et qu'on observe passer 40 minutes à chercher la bonne information dans 12 endroits différents. C'est exactement le cas où un agent peut apporter quelque chose de mesurable.

Temps 3 · Tester avec de vrais utilisateurs

Une fois le premier mc-file fabriqué (selon la méthode du chapitre 4), vous le testez avec un utilisateur réel du métier, pas avec vous-même. C'est probablement l'étape la plus difficile à organiser et la plus importante.

J'utilise un pattern emprunté à l'édition : le **beta reader**. Une personne, choisie pour sa connaissance du métier et sa capacité à formuler ses critiques, qui utilise l'agent comme elle l'utiliserait dans son travail réel. Elle me dit ce qui marche, ce qui ne marche pas, ce qui sonne faux, ce qui manque.

Les beta readers les plus utiles ne sont pas les plus favorables. Cherchez quelqu'un d'exigeant, qui n'a pas peur de dire « ça ne va pas du tout » ou « cet agent ne comprend rien à mon métier ». C'est de leurs critiques que vient la vraie valeur.

Trois ou quatre beta readers par mc-file suffisent en général. Au-delà, vous obtenez les mêmes retours et vous perdez du temps.

Temps 4 · Itérer en présence des utilisateurs

Les retours des beta readers ne sont pas une liste de bugs à corriger. Ce sont des indices sur ce que le métier attend vraiment. Vous reformulez le mc-file, vous le repassez en test, vous écoutez la nouvelle réaction.

Je donne en général deux à trois cycles d'itération avant de considérer un mc-file comme stable pour ce client. Plus vite que ça, vous avez sans doute oublié des cas. Plus lentement, vous êtes en train de polir au lieu de livrer.

Important : les itérations se font **avec** le beta reader, en sa présence, au-delà du simple traitement de ses retours. Quand c'est possible, on regarde ensemble la sortie de l'agent, on en discute, on décide la modification, je la fais en direct. Ça construit la confiance et ça transmet du savoir au passage.

Séparer la couche commune de la couche client

Quand vous travaillez sur plusieurs clients du même secteur, vous voyez vite ce qui revient (les mêmes types de mc-files, les mêmes contrats d'interface, les mêmes familles de décisions) et ce qui ne revient pas (le vocabulaire, les règles internes, les seuils, les rôles).

Je structure mes livrables en deux couches.

Une couche commune, que je maintiens entre les missions : les mc-files génériques du secteur, les structures de manifestes, les patterns de communication entre agents. Cette couche est mon capital. Plus j'enchaîne les missions, plus elle s'enrichit.

Une couche client, construite à chaque mission : le vocabulaire propre, les seuils propres, les règles propres, les périmètres d'agents adaptés au cas du client.

La couche client lit la couche commune (elle ne la duplique pas). Quand j'améliore la couche commune sur une mission, le bénéfice peut profiter aux clients précédents si c'est bien isolé.

C'est ce qui permet de fabriquer plus vite à chaque nouvelle mission, sans tomber dans le kit générique.

Ce qu'il faut négocier en amont

Quelques points à régler avec le client avant de commencer. Ils font la différence entre une mission qui se passe bien et une mission qui s'enlise.

L'accès aux utilisateurs réels. Si on vous propose seulement des entretiens avec le commanditaire et personne d'autre, refusez ou renégociez. Vous ne ferez pas un livrable utile.

Le temps des beta readers. Identifier les bonnes personnes, leur réserver une heure ou deux par semaine pendant la phase de test. C'est à mettre dans le devis et à mettre dans le planning.

L'autorité de décision. Qui tranche quand un beta reader veut une chose et un autre veut le contraire ? Définissez la chaîne de décision avant le démarrage. Sinon vous vous retrouvez à arbitrer en aveugle.

La maintenance après livraison. Un système agentique livré et oublié dérive en quelques mois (les modèles changent, les besoins évoluent, les agents accumulent du bruit dans leur état). Cadrez cette suite dès le départ, même si elle sera réalisée par le client en interne.

Anti-patterns

Construire un agent à partir de votre intuition du métier. Si vous n'avez pas interviewé, pas observé, pas testé, ce que vous fabriquez sera plausible et inutile.

Beta reader unique et favorable. Vous n'apprenez rien si votre testeur est d'accord avec tout. Cherchez la critique.

Itération sans fin. Vous polissez parce que vous avez peur de livrer. Au troisième cycle, livrez et passez à la suite. Le système continuera d'évoluer, c'est normal.

Promesse de livrer un système clé en main qui ne demande plus rien. C'est un mensonge, même bien intentionné. Tout système agentique demande de la maintenance et le client doit le savoir.

Confondre faire pour soi et faire pour eux. Vous savez ce que vous voudriez si c'était votre métier. Ce n'est pas leur métier. Méfiez-vous des choix qui sont juste les vôtres projetés sur eux.

Récapitulatif de la Partie II

Ce chapitre clôt la Partie II. Vous savez maintenant fabriquer des mc-files en méthode, les faire communiquer dans un système cohérent et porter cette méthode chez un client dont vous ne maîtrisez pas le métier.

C'est l'arsenal du constructeur de systèmes agentiques. La Partie III qui suit raconte une mise en application concrète : la fabrication du starter kit Scanderia, depuis la recherche initiale jusqu'au produit livré dans la masterclass.

Cette partie ouvre un cas d'étude plutôt qu'une recette à reproduire, avec ses choix et ses abandons, pour que vous puissiez vous en servir comme miroir quand vous construirez le vôtre.

Chapitre 7

Comment on a conçu le starter kit Scanderia

La Partie III ouvre sur un cas concret : la fabrication du starter kit qui sert de support à la masterclass. L'objectif n'est pas de vous faire reproduire notre kit. C'est de vous montrer un processus de conception réel, avec ses arbitrages, ses itérations et ses abandons, pour que vous puissiez vous en servir comme miroir.

La Partie III ouvre sur un cas concret : la fabrication du starter kit que vous avez vu mentionné dans les livrets 1 et 2, qui sert de support à la masterclass.

Vos clients et votre métier diffèrent des nôtres. L'objectif de ce chapitre n'est pas de vous faire reproduire notre kit. L'idée est plutôt de vous montrer un processus de conception réel, avec ses arbitrages, ses itérations et ses abandons, pour que vous puissiez vous en servir comme miroir quand vous construirez votre propre kit.

Je vais raconter sans embellir. Quand un choix s'est révélé mauvais, je le dis. Quand un choix tient bien, je dis pourquoi. Quand un choix reste ouvert dans ma tête, je dis ça aussi.

Le point de départ

Le starter kit n'a pas été conçu en chambre. Il est sorti d'une contrainte : préparer une masterclass dans laquelle les participants devaient repartir avec quelque chose qui marche chez eux dès le lendemain.

Cette contrainte a fixé trois propriétés non négociables.

Le kit devait tourner sans installation lourde. Pas de serveur à monter, pas de compte à créer ailleurs que chez Anthropic, pas de configuration système avancée. Quelqu'un qui sortait de la masterclass devait pouvoir l'utiliser depuis son Mac dans la journée.

Le kit devait être lisible par un humain non technique. Le public de la masterclass était mixte (entrepreneurs, indépendants, dirigeants, quelques développeurs). Les fichiers devaient pouvoir être ouverts, lus et modifiés sans compétences de codage.

Le kit devait être suffisamment générique pour servir d'amorce à plusieurs métiers. Sans tomber dans le piège du chapitre 6 (le kit générique qui ne sert à personne).

Ces trois contraintes ont éliminé une bonne partie des designs théoriquement possibles. Ce qui restait à décider après ça ressemblait plus à une optimisation sous contraintes qu'à une

page blanche.

Pourquoi 20 rôles dans le kit

La question revient à chaque fois : pourquoi avoir choisi exactement 20 agents dans le kit ?

La réponse honnête : c'est le résultat d'un raisonnement en deux étapes, sans optimum mathématique sous-jacent.

Première étape : le nombre minimal pour que la promesse tienne. La promesse de la masterclass était d'aider chacun à constituer une équipe d'agents complète, comme s'il recrutait. Une équipe complète, dans un cabinet ou une petite structure, ce n'est pas trois personnes. C'est entre 10 et 30, selon la taille et la diversité des fonctions. En dessous de 10 rôles, on restait au stade de la démo, sans atteindre celui d'une équipe.

Deuxième étape : le nombre maximal pour que ça reste appropriable. Au-delà de 25 ou 30 rôles, le manifeste devient impossible à tenir en tête, le découpage des responsabilités devient flou et l'utilisateur se perd. On a testé une version à 35 rôles : impossible à présenter en masterclass, impossible à mémoriser pour l'utilisateur final. Abandonnée.

20 est tombé entre les deux. Assez pour couvrir un éventail réaliste de fonctions (commercial, support, communication, veille, administratif, juridique, RH, finances, opérations, production), pas trop pour qu'on puisse encore tous les nommer par cœur après une heure de prise en main.

Il n'y a rien de magique dans ce nombre. Pour votre client, ce sera peut-être 8, peut-être 30. La méthode est : compter le minimum pour tenir la promesse, le maximum pour rester appropriable et choisir entre les deux.

Comment on a découpé les fichiers

Le starter kit ne contient pas un fichier par rôle. Il en contient plusieurs par rôle. Ce choix vient d'une itération qui mérite d'être racontée.

Première version : un fichier par rôle. Chaque rôle avait un seul mc-file, qui contenait tout (header de déclenchement, overview, workflow, format de sortie, exemples, documentation). Avantage : simple à présenter, simple à trouver. Inconvénient découvert au bout de deux semaines de test : les fichiers devenaient longs (4 à 6 pages chacun), les agents avaient du mal à se concentrer sur l'essentiel et la moindre modification demandait de relire le fichier entier.

Deuxième version : un fichier d'instruction principal + fichiers annexes par fonction. On a séparé le mc-file principal (court, ciblé) des fichiers complémentaires (workflows détaillés, exemples étendus, vocabulaire spécifique). L'agent charge le principal en permanence et va chercher les annexes seulement quand il en a besoin.

Cette séparation a réglé deux problèmes d'un coup. Les fichiers principaux sont restés courts (1 à 2 pages), lisibles d'un coup d'œil. Les annexes ont pu s'enrichir sans encombrer le contexte.

Troisième couche, ajoutée plus tard : la bible commune. Un ensemble de fichiers que tous les agents lisent (lexique général, ton, règles d'escalade, identité de l'équipe humaine qui les a déployés). Décrite dans le livret intermédiaire, chapitre 12. Cette couche n'était pas dans la version initiale du kit. Elle est apparue au moment où on a vu que sans elle, les agents prenaient des décisions cohérentes en silo, mais incohérentes entre eux.

Le découpage final tient sur trois niveaux : bible commune (partagée par tous), mc-file principal par rôle (ciblé, court), annexes (chargées à la demande).

Ce qu'on a essayé et abandonné

Quelques décisions qui ne sont pas dans le kit final. Voici pourquoi.

Un agent meta qui choisit lui-même quel agent appeler. Idée séduisante : un seul point d'entrée, l'utilisateur formule sa demande, l'agent meta route vers le bon spécialiste. On l'a implémenté, testé, abandonné. Pourquoi : le routage automatique masque l'équipe à l'utilisateur et le but de la masterclass était au contraire de lui faire prendre conscience de son équipe d'agents. Le kit final laisse l'utilisateur appeler explicitement chaque agent, ce qui est plus pédagogique.

Une mémoire centralisée à laquelle tous les agents accèdent. Idée tentante : une grande base partagée où chacun puise. On l'a écartée pour deux raisons. La première : ça demandait une infra plus complexe (chapitre 2), incompatible avec le « tourne sans installation ». La seconde : on a constaté que les agents fonctionnent souvent mieux avec une mémoire locale par rôle, qu'ils synthétisent ensuite par le chef de bureau. Le détour par la synthèse est moins efficace en apparence, mais beaucoup plus auditable et plus robuste.

Des agents totalement autonomes (qui décident sans validation humaine sur des actions externes). Tentés au début par le côté spectaculaire. Reculés rapidement après quelques cas où l'agent a fait quelque chose de bien intentionné mais inadapté. Le kit livré demande systématiquement validation humaine sur les actions à effet externe (envoi d'email, publication, paiement). C'est moins impressionnant en démo. C'est viable en production.

Les choix de formulation

Une part importante du travail a porté sur la façon dont les mc-files sont écrits, indépendamment de leur contenu.

Quelques principes qu'on a fixés progressivement.

Pas de jargon technique dans les mc-files lus par les agents métier. Mots comme « parser »,

« endpoint », « token » remplacés par leurs équivalents accessibles, ou expliqués au moment où ils apparaissent. Cette règle est venue d'une relecture critique : un beta reader non technique a buté sur plusieurs termes qu'on trouvait évidents. On a tout reformulé.

Le « vous » plutôt que le « tu ». Choix de cohérence avec les livrets et avec le ton Scanderia. L'intention est d'assumer un cadre professionnel, sans rechercher la formalité pour elle-même.

Pas de présupposé sur l'humilité ou la compétence du lecteur. On évite les formulations comme « si vous êtes débutant » ou « vous comprendrez vite ». Les agents s'adressent à l'utilisateur sans le hiérarchiser.

Une phrase d'identité par agent, en haut du mc-file. « Je suis [rôle], je m'occupe de [périmètre]. » Cette phrase est destinée d'abord à l'humain qui ouvre le fichier et veut comprendre en deux secondes à qui il a affaire, plus qu'à l'agent lui-même.

La validation contradictoire

On n'a pas livré le kit sans le confronter à des relecteurs qui n'avaient pas participé à sa fabrication.

Trois lectures critiques nous ont fait revoir des choix.

Une relectrice a souligné que le kit s'adressait implicitement à des indépendants, alors que la moitié du public masterclass travaillait en entreprise. On a ajouté un cadrage en préface et adapté plusieurs exemples.

Une autre a relevé que certains agents tombaient dans le jargon tech malgré nos efforts (« sandbox », « parser »). On a revu la liste des termes interdits.

Une troisième a noté que le kit présupposait un rythme de travail qui ne correspondait pas à toutes les structures (les agents tournaient le matin, supposant une journée de bureau classique). On a ajouté de la souplesse sur les déclencheurs.

Le kit livré est meilleur que celui qu'on aurait livré sans ces lectures. Plus juste, sans être nécessairement plus joli.

Ce qu'on referait différemment

Sur ce kit précis, deux choses nous accrochent encore.

Le découpage entre mc-file principal et annexes n'est pas homogène. Certains rôles ont une annexe, d'autres en ont quatre. Cette hétérogénéité reflète l'histoire de la fabrication plus qu'un design réfléchi. Pour une prochaine itération, on stabiliserait une grille (zéro, une ou deux annexes par rôle, jamais plus).

Le manifeste équipe n'a pas tout à fait le bon niveau d'abstraction. Il décrit qui fait quoi, mais

ne décrit pas les liens entre les rôles aussi précisément qu'il pourrait. Un nouvel utilisateur comprend la liste, mais pas tout de suite la chorégraphie. C'est un sujet sur lequel on travaille pour la version suivante.

Aucun de ces deux points n'est bloquant pour la valeur du kit. Ils sont de l'ordre du raffinement. Mais ils sont là et il faut savoir le voir pour pouvoir améliorer.

Récapitulatif avant de tourner la page

Le starter kit Scanderia a été conçu sous trois contraintes fortes (pas d'installation, lisible par non-tech, suffisamment générique). Le nombre de rôles a été choisi entre un minimum (crédibilité de l'équipe) et un maximum (appropriabilité). Le découpage des fichiers a évolué en trois temps, depuis « un fichier par rôle » jusqu'à la structure à trois niveaux actuelle. Plusieurs idées tentantes ont été abandonnées (agent meta, mémoire centralisée, agents totalement autonomes) parce qu'elles cassaient une contrainte.

Le chapitre suivant, dernier du livret, raconte la méthode de recherche qui a alimenté tout ça : comment on a su quoi mettre dans le kit, quelles sources on a utilisées et comment on a vérifié nos propres claims avant de les enseigner.

Chapitre 8

La méthode de recherche derrière le kit

Le chapitre précédent a montré comment on a fabriqué le starter kit. Celui-ci traite de l'amont : comment on a su quoi mettre dedans. Trois sources qui s'alimentent (veille produit, veille communautaire, demandes du marché), un processus en trois temps pour analyser cent vingt questions, une discipline de validation contradictoire. C'est aussi la conclusion du livret.

Le chapitre précédent a montré comment on a fabriqué le starter kit. Celui-ci traite de l'amont : comment on a su quoi mettre dedans. C'est aussi la conclusion du livret.

La question est moins évidente qu'il n'y paraît. Quand vous décidez quels agents inclure dans un kit, quels workflows encoder, quels exemples utiliser, vous faites des choix qui orientent ce que vos utilisateurs penseront possible avec l'IA. Ces choix méritent d'être informés par autre chose que votre intuition.

Ce chapitre décrit le processus de recherche que j'utilise. Il est généralisable : la même démarche sert à concevoir un kit pour un client dans un secteur que vous ne connaissez pas (chapitre 6).

Trois sources qui s'alimentent mutuellement

Je travaille à partir de trois flux d'information distincts, qui se complètent sans se substituer.

La veille produit. Tout ce que les fournisseurs (Anthropic, OpenAI, Google, Mistral, Meta) annoncent : nouveaux modèles, nouvelles fonctionnalités, changements de tarification, limites mises à jour. Cette veille est facile à organiser parce que les sources sont identifiées (blogs officiels, pages de release notes, comptes X des équipes produit). Elle est indispensable parce que le terrain bouge tous les mois. Un kit conçu sur les capacités d'il y a six mois sera obsolète sur plusieurs points.

La veille communautaire. Ce qui se dit dans les milieux techniques avancés : papiers de recherche, dépôts open source qui montent, threads de discussion, retours d'expérience de constructeurs en première ligne. Cette veille est plus exigeante (le bruit est élevé, le signal dispersé) mais c'est là qu'on voit arriver les patterns qui deviendront standards six mois plus tard. Pour le kit Scanderia, plusieurs choix de design viennent de papiers ou de fils de discussion qui n'avaient pas encore atteint la presse spécialisée.

Les demandes du marché. Ce que les utilisateurs réels demandent, butent contre, n'arrivent pas à faire. Pour notre kit, ça vient principalement de trois canaux : les questions des participants des masterclass précédentes, les échanges avec les prospects en avant-vente, les retours des clients en mission. Cette source est la plus difficile à analyser parce qu'elle est qualitative, mais c'est celle qui dit le plus clairement quel kit sera utile.

Aucune des trois sources, prise seule, ne suffit. La veille produit dit ce qu'on peut faire. La veille communautaire dit ce qui va émerger. Les demandes du marché disent ce qui sera adopté.

Comment on a analysé les questions des participants

Avant la masterclass, on disposait d'environ cent vingt questions reçues lors des sessions précédentes et des échanges en avant-vente. Plutôt que de les traiter une par une, on les a passées dans un processus en trois temps.

Premier temps : classification. Chaque question rangée dans une catégorie thématique (technique, business, juridique, opérationnel, conceptuel). Cette première passe a fait émerger les zones où les gens butent vraiment et celles où on s'imaginait qu'ils butaient sans que ce soit le cas. Surprise notable : très peu de questions techniques pures et beaucoup plus de questions sur l'organisation et le périmètre qu'on ne s'y attendait.

Deuxième temps : regroupement. Plusieurs questions formulées différemment posaient la même question de fond. On a fusionné. Sur cent vingt questions de départ, on a retenu une quarantaine de besoins distincts.

Troisième temps : priorisation. Pour chaque besoin, on s'est demandé : est-ce qu'on peut y répondre dans le starter kit (en ajoutant un agent, un fichier, un exemple), dans la masterclass (en y consacrant une démonstration), ou dans le livret (en y consacrant un chapitre) ?

Cette priorisation a directement structuré la collection. Les besoins les plus fréquents sont devenus les chapitres centraux du livret débutant. Les besoins plus avancés sont remontés dans le livret intermédiaire et les besoins encore plus spécifiques (méthode, fabrication, design) ont alimenté ce livret-ci.

C'est un processus simple, qui ne demande pas d'outillage particulier. Un fichier de notes, quelques heures de travail patient, une vraie écoute de ce qui revient. La rigueur tient dans la discipline (ne pas survoler), bien plus que dans la sophistication de la méthode.

Validation contradictoire

Une fois les hypothèses faites et les premiers contenus écrits, il reste un risque sérieux : se convaincre soi-même. On a construit une thèse, on a sélectionné les éléments qui la confirment et on n'a pas vraiment cherché ceux qui la contredisent.

Le réflexe que j'applique systématiquement : pour chaque claim important du kit ou du livret, chercher activement ce qui pourrait l'infirmer.

Trois angles d'attaque que j'utilise.

Le contre-exemple. Pour chaque règle qu'on énonce, chercher au moins un cas où elle ne s'applique pas et décider si ce cas mérite une exception explicite ou si la règle reste valide malgré lui. Si le contre-exemple casse la règle, on reformule.

La lecture par un sceptique. Un beta reader choisi pour son exigence et son indépendance lit le contenu en cherchant les endroits qui sentent le marketing, l'approximation, la prise de raccourci. Trois lectures critiques nous ont fait modifier des passages substantiels avant publication.

La vérification factuelle des chiffres. Tout chiffre cité (prix, fenêtre de contexte, capacité) est vérifié dans la documentation officielle au moment de l'écriture et daté pour qu'on sache à quel moment cette vérification a été faite. Les chiffres qui n'ont pas pu être sourcés ont été soit qualifiés (« la majeure partie »), soit retirés.

Cette discipline coûte du temps. Pour le seul livret intermédiaire, plusieurs heures ont été passées à vérifier ou à reformuler des passages qui ne tenaient pas la lecture contradictoire. La conséquence : ce qui reste dans le livret peut être défendu publiquement.

Anti-patterns en recherche

Quelques pièges que je rencontre régulièrement chez les constructeurs.

Confondre veille et action. Lire des dizaines d'articles, suivre des dizaines de comptes, ne rien produire. La veille n'a de valeur que si elle alimente une décision concrète sur ce que vous fabriquez.

S'enfermer dans une seule source. Ne suivre que la veille produit (et rester en retard d'un an sur ce qui émerge), ou ne suivre que la veille communautaire (et passer à côté des limites réelles des outils disponibles aujourd'hui), ou n'écouter que le marché (et fabriquer ce que les gens demandaient hier).

Ignorer les questions qui ne rentrent pas dans la grille. Une ou deux questions étranges qui ne se classent nulle part, on les met de côté. Souvent, ce sont elles qui pointent vers le besoin qu'on n'avait pas vu venir. Gardez-les en suspens, relisez-les après quelques semaines.

Ne pas dater ses propres analyses. Une note rédigée il y a six mois sur les capacités des modèles peut être obsolète, mais si elle n'est pas datée, vous l'utiliserez encore aujourd'hui sans vous en méfier. Datez tout. Re-vérifiez avant publication.

Conclusion du livret

Vous arrivez au bout. Récapitulons ce qui a été dit.

La Partie I a posé la thèse. Un système agentique vivant tient sur trois briques : un état partagé que les agents lisent et modifient, des décisions encadrées prises sur cet état, des conversations qui rendent ces décisions traçables. Le pivot pédagogique est la décision : c'est ce qui distingue un système qui s'adapte d'un système qui se fige.

La Partie II a décrit l'atelier. Comment on fabrique un mc-file de qualité production en cinq étapes (spécifier, écrire, tester simple, itérer sur cas limites, stabiliser). Comment on fait communiquer plusieurs mc-files par passage de témoin, tableau commun ou requête, en respectant des contrats d'interface explicites. Comment on porte cette méthode chez un client dont on ne maîtrise pas le métier, en distinguant couche commune (votre capital) et couche client (construit ensemble).

La Partie III a ouvert le capot du starter kit Scanderia. Pourquoi vingt rôles. Comment on a découpé les fichiers. Quelles idées on a abandonnées. Quelle méthode de recherche a alimenté tout ça.

À ce point, vous avez les pièces et le mode d'emploi. Reste à construire. La construction prendra plus de temps que vous ne le pensez aujourd'hui et c'est normal. Le savoir-faire d'un constructeur de systèmes agentiques tient avant tout dans la patience qu'il met à spécifier, à tester, à corriger, à écouter ses utilisateurs réels et à reconnaître quand ses propres choix initiaux n'étaient pas les bons. La maîtrise des outils ne suffit pas.

Je continue d'apprendre. Si vous trouvez des choses que ce livret n'a pas vues, ou que vous voyez autrement, écrivez-moi. Le terrain bouge vite et personne ne le couvre tout seul.