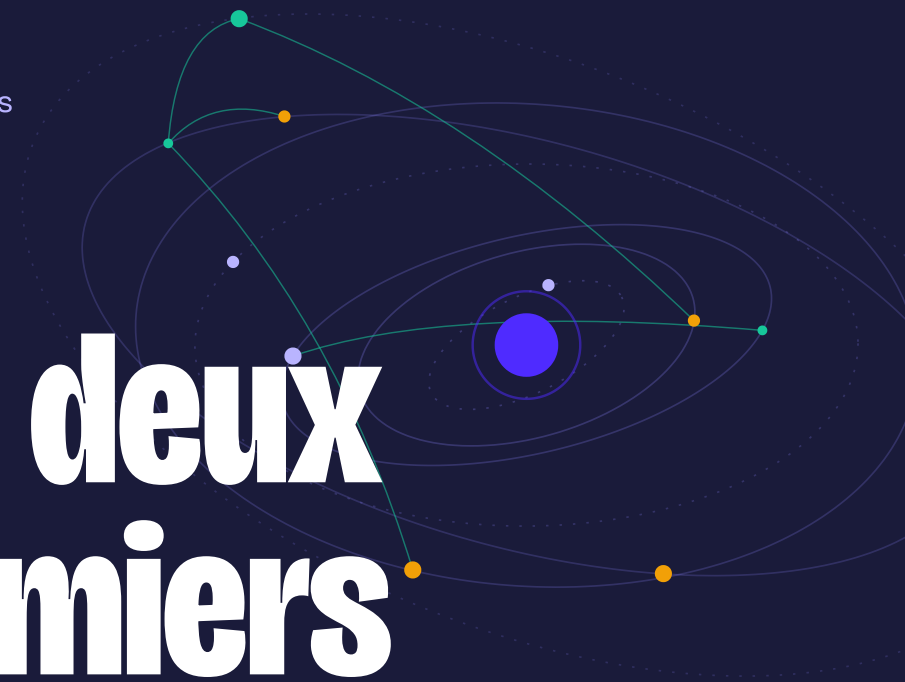
Chapitres gratuits



Les deux premiers chapitres

Ce qu'est vraiment un agent, et Les workflows avant les agents : les deux premiers chapitres d'Agents en production, le guide pratique des frameworks agentiques, de MCP, des skills et de l'automatisation par l'IA. Gratuits.

Écrit par des agents IA (Claude), sous la direction de Yann

Édition 2026 à 2027

Ce qu'est ce livre

Agents en production est un guide pratique pour faire passer des agents IA d'un pilote prometteur à un système qui tourne tous les jours et se rentabilise. Il est indépendant des éditeurs et construit autour d'un seul modèle mental : un agent, c'est un modèle, des outils, une boucle et du contexte, enveloppés dans un *harness* (le code qui encadre la boucle du modèle) qui applique budgets, garde-fous, état et télémétrie.

Il s'adresse à quatre lecteurs, plus un parcours Claude pour les équipes qui construisent sur Claude : Tech (développeurs et tech leads), Manager, Solo (fondateurs solo) et No-code (pros de l'automatisation sans code). Chaque chapitre indique les sections que votre parcours peut sauter.

Vous saurez :

- Décider quand un *workflow* (enchaînement d'étapes prédéfinies dans le code) fixe vaut mieux qu'un agent, et concevoir des outils, des serveurs MCP, des *skills* (dossiers avec un `SKILL.md`, chargés à la demande) et un contexte qu'un modèle peut exploiter.
- Choisir un framework, ou aucun, et envelopper la boucle dans un harness avec budgets, garde-fous, état et télémétrie.
- Exploiter des agents avec des *evals* (évaluations automatisées : jeux de tâches avec résultat attendu et notation), du traçage, le coût par succès, la sécurité et la gouvernance, y compris le règlement européen sur l'IA (AI Act).
- Mettre des agents au travail pour construire et tester du logiciel, avec un point de contrôle humain à chaque poste qui en a besoin.
- Déployer un agent, ou l'arrêter, sur la base de preuves qu'un comité de pilotage peut vérifier.

-

Écrit par des agents IA, à dessein

Ce livre a été écrit par une équipe d'agents IA. Claude, un modèle conçu par Anthropic, a orchestré plus de trente agents spécialisés qui ont mené les recherches, rédigé, vérifié auprès des sources primaires, dessiné les figures et construit les pages que vous lisez. Un humain, Yann, a dirigé le travail par de courtes décisions à choix multiples et répond de ce qui est publié. Cette édition française a elle aussi été écrite par des agents IA, sous la direction de Yann : des agents traducteurs l'ont produite à partir de l'édition anglaise. Un livre sur les agents en production est aussi un test des agents en production : si vous trouvez une erreur, nous voulons le savoir. La postface, « Comment ce livre a été fait », montre la chaîne de production, les chiffres, et les erreurs rattrapées ou peut-être manquées.

Comment il traite les faits et les histoires

Chaque affirmation précise (un chiffre, une date, une version, une échéance légale) renvoie à sa source, et le site web liste toutes les sources par chapitre ([SITE URL]). Le cahier d'exercices compagnon contient les exercices, les corrigés, le projet fil rouge et l'annexe Claude Kit. Les chiffres illustratifs sont signalés comme tels. Un conflit d'intérêts : les agents qui ont écrit ce livre tournent sur Claude, et le livre traite des produits, de la recherche et des données d'Anthropic au même titre que de ceux de tous les autres. Le chapitre 19, le seul chapitre propre à un éditeur, a été ajouté à la demande du directeur et obéit à la même règle de sourçage. Pour tous les éditeurs : des sources primaires, des chiffres rapportés par les éditeurs signalés comme tels, et aucune recommandation de produit sans règle de décision derrière. Les agents n'ont pas d'expérience vécue, et ce livre ne prétend donc pas en avoir : les histoires sont soit des incidents publics cités, soit des cas composites clairement signalés. Quand le directeur humain ajoute sa propre expérience de *product builder* (développeur polyvalent responsable d'un produit) travaillant sur une plateforme d'environ 100 microservices et un assistant RAG interne, elle apparaît dans un encadré « Note de Yann ». Cette édition en contient un, au chapitre 13, rédigé à partir de ses réponses à un entretien et approuvé par lui ; d'autres pourront venir dans les éditions suivantes.

Compléments en ligne

Gratuits sur le site web ([SITE URL]) :

- Notes de terrain 2026 : le recueil daté complet derrière le chapitre 20, avec une source et un niveau de preuve pour chaque entrée.
- L'annexe B, la checklist de conception des outils et serveurs MCP, et l'annexe C, le modèle de modélisation des menaces, chacune avec un PDF.
- Pour chaque chapitre : les questions des lecteurs, un quiz d'autoévaluation et toutes les sources.
- Le glossaire.

Comment cette édition reste à jour

C'est une édition vivante, avec des mises à jour gratuites jusqu'à fin 2027. Les affirmations qui évoluent vite sont marquées « (fin 2026) » pour que vous sachiez quoi revérifier.

Droits d'auteur et mentions

Agents en production : Le guide pratique des frameworks agentiques, de MCP, des skills et de l'automatisation par l'IA. Édition 2026 à 2027. Édition numérique, 2026.

Copyright © 2026 Yann. Tous droits réservés. Aucune partie de ce livre ne peut être reproduite, stockée ou transmise sous quelque forme que ce soit sans l'autorisation écrite de l'éditeur, à l'exception de brèves citations dans des critiques, ainsi que des checklists et des modèles, que les lecteurs peuvent copier pour un usage interne à leur propre organisation.

Paternité IA. Anthropic n'a ni commandé, ni relu, ni approuvé ce livre. La protection des textes générés par IA varie selon les juridictions ; les droits sont revendiqués sur la sélection, l'agencement et les contributions d'auteur humain.

Avertissement. Ce livre a une vocation d'information et de formation. Il ne constitue pas un conseil juridique, financier, fiscal, de sécurité ou autre conseil professionnel. Les faits, prix, versions et réglementations ont été vérifiés en novembre 2026 et évoluent vite ; vérifiez-les avant d'agir. L'auteur et l'éditeur déclinent toute responsabilité pour les décisions prises sur la base de ce livre.

Marques. Les noms de produits et d'entreprises (Claude, ChatGPT, Gemini, Copilot, n8n, Zapier, Make et tous les autres cités dans ce livre) sont des marques de leurs propriétaires. Ils ne sont utilisés que pour identifier des produits, et aucune affiliation ni approbation n'est sous-entendue.

ISBN (édition numérique) : à attribuer. Publié par [Publisher name].

Composé en Bricolage Grotesque, Source Serif 4 et JetBrains Mono, tous sous licence SIL Open Font License 1.1. Figures dessinées par l'équipe d'agents.

Vous avez trouvé une erreur ? Chaque correction améliore l'édition suivante : [errata contact to be added].

Les citations de sources anglophones sont traduites par nos soins ; les liens renvoient aux originaux.

1

Ce qu'est vraiment un agent

Une définition assez précise pour concevoir, budgéter et argumenter

Tech Manager Solo No-code Claude

Demandez à cinq personnes ce qu'est un « agent » et vous obtiendrez cinq réponses, chacune impliquant un budget, un profil de risque et une équipe différents.

Quatre éléments et un harness

Un agent est **un modèle qui tourne en boucle, choisit des outils, dispose d'un contexte géré, le tout à l'intérieur d'un *harness* (le code qui encadre la boucle du modèle : budget, garde-fous, état, télémétrie)**. Retirez l'un des quatre éléments et vous obtenez autre chose. Retirez le harness et vous obtenez une démo.

Le modèle

Le modèle est le décideur : à chaque tour, il lit le contexte et renvoie une réponse finale ou un appel d'outil (*tool call*). Il ne « se souvient » pas et n'« agit » pas ; c'est votre code qui le fait, quand le modèle le demande. Quand une exécution échoue, trouvez dans sa trace la première étape qui a dévié et posez, dans l'ordre, ces questions :

1. **L'information nécessaire était-elle dans le contexte ?** Sinon, défaillance de contexte : corrigez la récupération ou la gestion du contexte.
2. **Était-elle présente, mais le modèle a-t-il choisi le mauvais outil ou les mauvais arguments ?** Défaillance du modèle : corrigez les instructions, les descriptions d'outils ou le choix du modèle, et ajoutez le cas à votre jeu d'évaluation.

3. **L'appel était-il sensé, mais l'outil s'est-il mal comporté ?** Défaillance logicielle : corrigez le code, ajoutez un test unitaire.

Cela suppose que chaque appel au modèle et chaque appel d'outil soient tracés avec leurs entrées.

Choisissez le modèle comme une paire : modèle et effort de raisonnement. Les tableaux de lancement indiquent l'effort maximal ; les valeurs par défaut livrées sont souvent plus basses (le chapitre 19 en donne un exemple daté). Figez les deux valeurs dans la configuration et comparez les paires sur le coût par tâche réussie.

Les outils

Un outil, c'est un nom, une description, un schéma de paramètres et une fonction que le modèle peut vous demander d'exécuter. L'essentiel du risque est là : une recherche en lecture seule et un outil « rembourser le client » se ressemblent dans un schéma et ne sont pas du tout le même type d'objet (chapitre 3).

La boucle

Envoyez le contexte au modèle ; s'il renvoie un appel d'outil, validez-le, exécutez-le, ajoutez le résultat ; s'il renvoie une réponse finale ou si une condition d'arrêt est atteinte, sortez ; sinon, recommencez. On appelle généralement cette boucle ReAct, d'après l'article de 2022, et c'est la boucle par défaut de la plupart des SDK.

Le contexte

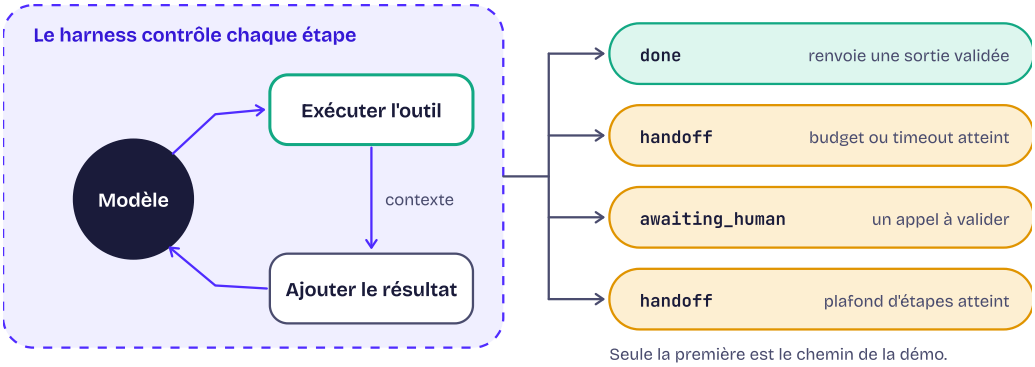
Le contexte, c'est tout ce que le modèle voit à un tour donné : instructions, définitions d'outils, documents récupérés, historique, résultats d'outils, mémoire. Le rappel se dégrade à mesure qu'il grossit, un phénomène que les travaux de Chroma ont baptisé ***context rot*** (dégradation du rappel et du respect des consignes quand le contexte se remplit) (Chroma, juillet 2025) ; le maîtriser relève de ce qu'Anthropic appelle l'ingénierie du contexte (*context engineering*) (Anthropic, 29 septembre 2025), si bien qu'une boucle qui ajoute

indéfiniment chaque résultat se dégrade au fil de l'exécution. La mémoire n'est pas un cinquième élément : c'est une source de contexte, lue par la boucle et écrite au moyen d'outils (chapitre 5).

Le harness

Trois statuts, quatre façons de terminer une exécution

Une démo gère une sortie. Un agent en production traite les quatre comme des issues normales.



Seule la première est le chemin de la démo.

Trois statuts, quatre sorties. Terminé est le chemin de la démo ; un passage de relais (budget, timeout ou plafond d'étapes) et l'attente d'un humain sont des issues normales que le harness doit gérer.

> **Retirez le harness et vous obtenez une démo.**

Le harness est l'enveloppe de production, avec quatre fonctions :

- **Budget** : plafonds sur les étapes, les *tokens* (unités de texte facturées par le modèle), l'argent et le temps réel écoulé.
- **Garde-fous** (*guardrails*) : outils autorisés, outils soumis à approbation humaine, entrées et sorties bloquées.
- **État** : points de reprise pour qu'une exécution reprenne après un crash, et attentes durables pour les approbations.
- **Télémétrie** : traces de chaque appel au modèle et de chaque appel d'outil, les données sur lesquelles tournent vos *evals* (évaluations automatisées : jeux de tâches avec résultat attendu et notation).

Les valeurs par défaut des frameworks sont un point de départ, pas une conception (chapitre 6). Voici l'ensemble en une esquisse ; `call_model` et `run_tool` remplacent votre client de modèle et votre registre d'outils. C'est la forme générale ; le chapitre 8 donne la version à copier.

```
import time

MAX_STEPS, MAX_COST_USD, TIMEOUT_S = 10, 0.50, 120

def run_agent(task: str, tools: dict, ctx: list) → dict:
    ctx.append({"role": "user", "content": task})
    cost, start = 0.0, time.monotonic()
    for step in range(MAX_STEPS):
        reply, call_cost = call_model(ctx, tools)
        cost += call_cost
        ctx.append(reply.as_message())
        trace("model_call", step=step, cost=call_cost)
        # fini seulement sur un submit_* validé (chapitre 8)
        if reply.submit:
            return {"status": "done", "output": reply.submit}
        if cost > MAX_COST_USD:
            return {"status": "handoff", "reason": "budget"}
        if time.monotonic() - start > TIMEOUT_S:
            return {"status": "handoff", "reason": "timeout"}
        if not reply.tool_calls:  # texte libre : pas fini
            return {"status": "handoff", "reason": "no submit"}
        # une approbation met tout le tour en pause (chapitre 8)
        pending = [c for c in reply.tool_calls
                   if needs_approval(tools, c)]
        if pending:
            checkpoint(ctx)
            return {"status": "awaiting_human", "pending": pending}
        for call in reply.tool_calls:
            tool = tools.get(call.name)
            if tool is None:
                ctx.append(tool_error(call.id, "outil inconnu"))
                continue
            ctx.append(safe_run(tool, call))
        checkpoint(ctx)
    return {"status": "handoff", "reason": "step cap"}

def safe_run(tool, call):
    try:
        return tool_result(call.id, run_tool(tool, call.args))
    except Exception as e:
        return tool_error(call.id, f"{type(e).__name__}: {e}")
```

Une exécution se termine par l'un de trois statuts : terminée (`done`), passage de relais (`handoff`) avec sa raison (budget, timeout, no submit, step cap), ou en attente d'un humain (`awaiting_human`). Une démo n'en connaît qu'un. Un agent de production traite les deux autres comme des issues normales. En attente d'un humain, persistez le point de reprise et les appels en attente sous un identifiant d'exécution ; pour les longues attentes, utilisez l'exécution durable (*durable execution*) plutôt qu'un processus bloqué (chapitre 7).

Anti-pattern

Le harness après coup

Le pilote typique n'a ni plafond d'étapes, ni plafond de coût, ni traces, « jusqu'au lancement ». D'ici là, les *prompts* (instructions envoyées au modèle) ont été réglés sur des exécutions sans limite, et un plafond de 10 étapes casse des tâches qui en prenaient discrètement 25. Mettez le harness dans le premier commit avec des valeurs généreuses, puis resserrez-les avec les données.

Choisir les premières valeurs

1. **Commencez large** : un plafond d'étapes d'environ trois fois ce dont une bonne exécution a besoin.
2. **Lancez votre jeu d'évaluation** : 30 à 50 tâches réelles ; notez étapes, coût et résultat.
3. **Fixez le plafond d'étapes à partir des seules exécutions réussies**, juste au-dessus de la plus lente (vers le 95e centile). Les exécutions plus longues sont généralement des boucles ou des tâches pour un humain.
4. **Déduisez le plafond de coût de la valeur** : une tentative doit coûter nettement moins que ce que vaut un succès (chapitre 14).
5. **Fixez le *timeout* (délai d'expiration) d'après l'appelant** : quelques secondes pour un chat, quelques minutes pour une tâche de fond.

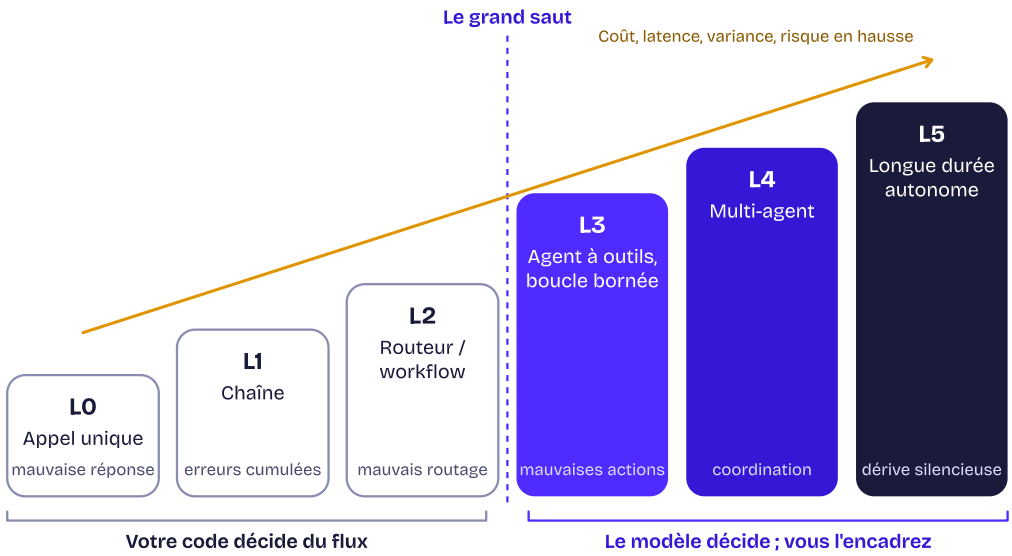
Un plafond par exécution arrête une exécution, pas un bug qui en déclenche des milliers ; ajoutez aussi une limite de dépense quotidienne et des limites de débit par utilisateur (chapitres 8 et 10).

L'échelle d'autonomie

Chaque échelon gagné ajoute des capacités, mais aussi du coût, de la latence, de la variance et du risque.

L'échelle d'autonomie

Chaque échelon ajoute des capacités, mais aussi du coût, de la latence, de la variance et du risque



> **Montez un échelon à la fois.** Sous L3, votre code décide du flux ; à partir de L3, le modèle décide et vous ne pouvez que le borner.

Niveau	Nom	Qui décide de l'étape suivante	Appels au modèle par tâche	Exemple	Risque principal
L0	Appel unique	Votre code	1	Classer un ticket	Mauvaise réponse sur une entrée
L1	Chaîne	Votre code, séquence fixe	Fixe	Extraire, rédiger, vérifier le ton	Les erreurs s'additionnent
L2	Routeur / *workflow* (enchaînement d'étapes prédéfinies dans le code)	Votre code, le modèle choisit les branches	Faible, connu par branche	Router un ticket vers la facturation, la technique ou les ventes	Mauvais routage

Niveau	Nom	Qui décide de l'étape suivante	Appels au modèle par tâche	Exemple	Risque principal
L3	Agent avec outils, boucle bornée	Le modèle, dans vos plafonds	Variable, jusqu'au plafond	Instruire un ticket avec la recherche et le CRM, 10 étapes au plus	Mauvaises actions, boucles, pics de coût
L4	Multi-agent	Plusieurs modèles, qui délèguent	Somme de toutes les boucles	Un orchestrateur répartit une recherche entre des exécutants	Défaillances de coordination
L5	Autonome de longue durée	Le modèle, sur des heures ou des jours	Illimité	Agent qui traite une file de tâches en attente	Dérive, dépense non bornée

Jusqu'à L2, vous pouvez chiffrer une tâche avant de livrer. À partir de L3, vous ne connaissez que le plafond, et comme chaque étape renvoie le contexte qui grossit, le coût augmente plus vite que le nombre d'étapes.

- **Montez un échelon à la fois.** Un routeur L2 bien construit bat un agent L3 fragile sur presque tous les indicateurs qui comptent pour la direction financière.
- **Le passage de L2 à L3 est le grand saut.** Sous L3, votre code décide du flux de contrôle et vous pouvez le dessiner. À partir de L3, le modèle décide et vous ne pouvez que le borner : il vous faut des evals de trajectoire, et le harness cesse d'être facultatif.
- **L4 et L5 multiplient tout** : les appels, le contexte et les endroits où les erreurs peuvent se cacher (chapitre 2).

Règle de décision

Par quel échelon commencer

Commencez au niveau le plus bas qui peut plausiblement faire la tâche. Des étapes que vous pouvez écrire à l'avance : L1 ou L2. Des étapes qui dépendent de ce que vous découvrez : L3, avec un plafond d'étapes que vous savez justifier. Ne commencez jamais en L4 ou L5 ; laissez les échecs de L3 vous dire ce que des agents supplémentaires devraient corriger.

Où se situent les déploiements réels

À fin 2026, la donnée publique la plus utile est l'étude menée par l'université de Berkeley « Measuring Agents in Production » (MAP) : 20 études de cas plus une enquête auprès de 306 praticiens, dont 86 ont rendu compte d'agents en production ou en pilote (arXiv).

- **Laisse courte (enquête).** 68 % des agents déployés exécutent 10 étapes ou moins avant qu'un humain intervienne ; 47 % moins de 5.
- **Les humains jugent (enquête, 31 répondants à cette question).** 74 % s'appuient principalement sur l'évaluation humaine ; 52 % utilisent un LLM juge (*LLM-as-judge*).
- **Code sur mesure (études de cas).** 17 équipes sur 20 ont construit sur des appels directs à l'API du modèle, alors que 61 % des répondants à l'enquête utilisent un framework.
- **Mesure faible (études de cas).** 75 % évaluent sans jeux de référence formels.

La plupart des agents en production sont des workflows L2 ou des agents L3 étroitement plafonnés, avec un humain à proximité. Une seule des 20 études de cas laissait son agent explorer sans contrainte, dans un bac à sable (*sandbox*) avec des contrôles de CI stricts. Si un premier projet promet du L5, demandez la version L3 avec un plafond d'étapes, une file de relecture et une date pour les résultats d'évaluation.

Agent, automatisation, chatbot, copilote : le vocabulaire qui clôt les réunions

Le critère : qui décide de l'étape suivante, et qui agit.

Comment nous en sommes arrivés là

Les agents ont été définis bien avant les LLM

Le mot « agent » a une histoire dans la recherche plus ancienne que la plupart des frameworks actuels. En 1995, Michael Wooldridge et Nicholas Jennings ont proposé une « notion faible d'agentivité » à quatre propriétés : l'autonomie (agir sans intervention humaine directe), la capacité sociale, la réactivité à un environnement changeant et la proactivité, c'est-à-dire un comportement orienté vers un but, à l'initiative de l'agent (Wooldridge et Jennings, 1995). Ce qui a changé après 2023, c'est qu'un modèle de langage généraliste peut tenir le rôle de décideur, si bien que l'ancienne définition décrit enfin un logiciel que les équipes peuvent livrer.

Leçon pour les agents : définissez l'agentivité par un comportement observable (autonomie, réaction, initiative), et non par le modèle qui se trouve à l'intérieur.

Terme	Qui décide de l'étape suivante	Qui agit	Exemple
Automatisation	Votre code (règles ou logigramme)	Votre code	Un workflow avec une étape LLM qui classe les e-mails
Chatbot	L'utilisateur, tour par tour	Le modèle répond ; rien d'autre ne se passe	Assistant FAQ sur une documentation
Copilote	L'utilisateur, avec des suggestions du modèle	L'utilisateur accepte ou rejette	Un brouillon de réponse que l'agent du support modifie
Agent	Le modèle, dans des limites	Le modèle, via des outils, avec des approbations	Instruit un ticket et propose une résolution

- Une automatisation qui contient un appel à un LLM reste une automatisation. L'appeler agent augmente les attentes et baisse la rigueur.
- Un chatbot devient un agent dès qu'il peut appeler des outils qui modifient quelque chose, ou enchaîner lui-même des appels d'outils.

- Un copilote est souvent la meilleure première version d'un agent : chaque acceptation ou rejet devient une donnée d'évaluation.
- « Autonome » est un échelon, pas un type. Dites « L3 avec approbation sur les écritures ». Un nœud « AI agent » dans n8n ou Zapier qui boucle sur des outils est du L3, quelle que soit l'allure du canevas (chapitre 13).

Cas composite : l'assistant devenu agent

Cas réaliste reconstitué, non observé. Un éditeur de logiciels de taille moyenne a construit un assistant documentaire interne : récupération fixe, réponses sourcées, niveau L1. Des questions comme « pourquoi l'import de ce client a-t-il échoué hier » ont conduit l'équipe à ajouter des outils un par un : une deuxième source documentaire, une recherche du responsable, une requête en lecture seule sur l'état des pipelines. Après le troisième, le système choisissait quels outils appeler et dans quel ordre.

Personne n'avait décidé de construire un agent ; l'équipe en avait un quand même, avec le harness d'un chatbot. Une question qui déclenchait une longue chaîne de recherches reformulées était lente, chère et invisible, jusqu'à ce que chaque étape soit tracée. Les corrections : un plafond d'étapes, un budget par question, des appels d'outils tracés, des identifiants en lecture seule et un jeu d'évaluation constitué des questions qui avaient mal tourné.

En production **Repérez le moment où vous franchissez L2**

Le passage du chatbot à l'agent se fait rarement en revue de conception ; il se fait un outil à la fois. Faites du deuxième outil ajouté à un assistant le déclencheur d'une revue du harness : plafonds, traçage, périmètre des identifiants, jeu d'évaluation.

Checklist

- [ ] Chaque fonctionnalité d'IA a son niveau d'autonomie (L0 à L5) écrit noir sur blanc.

- [ ] Chaque agent a un plafond d'étapes, un plafond de coût et un timeout dans sa configuration.
- [ ] Chaque agent gère trois statuts : terminé, passage de relais avec une raison, en attente d'un humain.
- [ ] Chaque appel au modèle et chaque appel d'outil sont tracés, pas seulement la réponse finale.
- [ ] Les outils d'écriture exigent une approbation, ou une raison documentée justifie de s'en passer.
- [ ] L'ajout d'un outil à un assistant existant déclenche une revue du harness.

Le chapitre 1 en une page

Ce qu'est vraiment un agent

Un agent est un modèle dans une boucle, qui choisit des outils, avec un contexte géré, à l'intérieur d'un harness. Nommez son échelon sur l'échelle avant de le construire.

À retenir

- C'est le harness (budget, garde-fous, état, télémétrie) qui rend un agent prêt pour la production
- Le passage de L2 à L3 est le grand saut : le modèle prend le contrôle du flux
- Les déploiements réels sont courts et supervisés ; les systèmes deviennent des agents un outil à la fois

À faire dès lundi

- Récupérez la trace d'une exécution échouée et trouvez la première étape qui a dévié
- Comptez les étapes de vos 30 dernières exécutions réussies et fixez le plafond juste au-dessus de la plus lente
- Demandez combien d'outils compte aujourd'hui votre plus ancien assistant, et qui a relu le dernier ajouté

68 %

des agents déployés dans l'enquête exécutent 10 étapes ou moins avant un humain

17 sur 20

équipes des études de cas ont construit sur des appels directs à l'API (61 % des répondants à l'enquête utilisent un framework)

2

Les workflows avant les agents

L'agent le moins cher et le plus fiable est souvent celui que vous n'avez pas construit

Tech | Manager | Solo | No-code | Claude

Cas composite (cas réaliste reconstitué, non observé) : un éditeur de logiciels passe deux semaines à construire quatre agents coopérants pour trier les tickets de support, à quarante secondes par ticket, jusqu'à ce que quelqu'un dessine le vrai processus sur un tableau blanc : cinq boîtes, deux branches. Reconstruit en un après-midi sous la forme d'un *workflow* (enchaînement d'étapes prédéfinies dans le code) avec deux appels au modèle, le système est plus rapide, moins cher et plus facile à tester.

Un **workflow** est un système où votre code décide de la séquence des étapes, même si certaines étapes appellent un modèle ; dans un agent (chapitre 1), c'est le modèle qui décide. La différence tient à qui contrôle le flux.

La règle de base : si vous pouvez dessiner le logigramme, construisez le workflow

Si vous pouvez dessiner le logigramme avant le début de la tâche, construisez le workflow. N'utilisez un agent que pour les parties où l'étape suivante dépend de ce que vous découvrez en chemin.

Comment nous en sommes arrivés là

Les workflows sont exécutables depuis cinquante ans

Composer de petites étapes prévisibles est une vieille idée. Les pipes Unix, attribués à Doug McIlroy et implémentés par Ken Thompson (Bell Labs, Unix history), apparaissent pour la première fois dans le manuel d'Unix Version 3 de février 1973 (Oral history of Unix). BPMN 2.0 a ensuite rendu les diagrammes de processus exécutables (prologue). Chaque génération a appris qu'un flux explicite est plus facile à tester, à auditer et à modifier qu'un comportement caché dans un composant astucieux.

Leçon pour les agents : un appel au modèle n'est qu'une étape de plus dans un pipeline ; gardez le pipeline explicite partout où vous pouvez le dessiner.

Dans un workflow, chaque chemin est du code que vous avez écrit : testable, journalisé, chiffré, explicable. Dans un agent, le modèle choisit le chemin à l'exécution : vous pouvez le borner, pas l'énumérer. Ce compromis n'est rentable que si vous avez besoin de cette flexibilité. L'article d'Anthropic « Building effective agents » trace la même ligne et conseille de commencer par la solution la plus simple, en n'ajoutant de la complexité qu'en cas de besoin (Anthropic, 19 décembre 2024). La plupart des « agents » en production sont des workflows avec une courte section agentique, et c'est la forme saine.

Pour vérifier que vous pouvez le dessiner, prenez cinq entrées réelles et notez les étapes qu'un humain compétent suit pour chacune. Mêmes étapes, ou deux ou trois variantes connues : un workflow. Un chemin différent à travers des systèmes différents à chaque fois : peut-être un agent.

Anti-pattern

Construire l'agent pour découvrir ce qu'est le workflow

Les équipes qui ne connaissent pas encore le processus construisent un agent en espérant que le modèle le trouvera. Il le trouvera, différemment à chaque fois, et vous passerez des semaines à rétro-analyser des traces. Interrogez l'humain qui fait la tâche aujourd'hui, dessinez le logigramme, construisez-le. Gardez l'agent pour les boîtes marquées « ça dépend ».

Le catalogue des patterns

Dans les quatre premiers patterns, votre code maîtrise le flux. Orchestrateur-exécutants se situe à la frontière. Les deux derniers sont des agents. Les cinq premières définitions suivent les recommandations d'Anthropic.

Pattern	Forme	À utiliser quand	À éviter quand	Coût relatif	Défaillance typique
Chaînage de prompts (*prompt chaining*)	Séquence fixe d'appels avec des contrôles en code entre eux	Étapes claires (extraire, rédiger, vérifier)	Les étapes dépendent les unes des autres de façon imprévisible	Faible	Une erreur précoce se propage aux étapes suivantes
Routage (*routing*)	Classer, puis envoyer vers un chemin spécialisé	Catégories d'entrées distinctes	Les catégories se recoupent ou changent sans cesse	Faible	Mauvais routage avec assurance
Parallélisation	Appels indépendants simultanés (découpage) ou même appel N fois (vote)	Sous-tâches indépendantes, jugements à fort enjeu	État partagé	Moyen, N fois les appels	Votes concordants sur la même mauvaise réponse
Évaluateur-optimiseur (*evaluator-optimizer*)	Générateur plus critique, en boucle	Critères de qualité vérifiables	Le premier jet suffit	Moyen à élevé	Peaufinage sans fin

Pattern	Forme	À utiliser quand	À éviter quand	Coût relatif	Défaillance typique
Orchestrateur-exécutants (*orchestrator-workers*)	Un modèle décompose et délègue	Sous-tâches inconnues à l'avance	Vous pouvez lister les sous-tâches	Élevé	Mauvaise décomposition
Boucle d'agent (ReAct)	Raisonner, appeler un outil, lire, recommencer	Le chemin dépend des résultats d'outils	Vous pouvez dessiner le logigramme	Borné seulement par les plafonds	Boucles, mauvais outils, paramètres inventés
Planifier puis exécuter (*plan-and-execute*)	Écrire un plan, l'exécuter, le réviser	Tâches longues où un plan visible aide la relecture	Tâches courtes	Moyen à élevé	Suivre fidèlement un mauvais plan

Ce que le tableau ne dit pas :

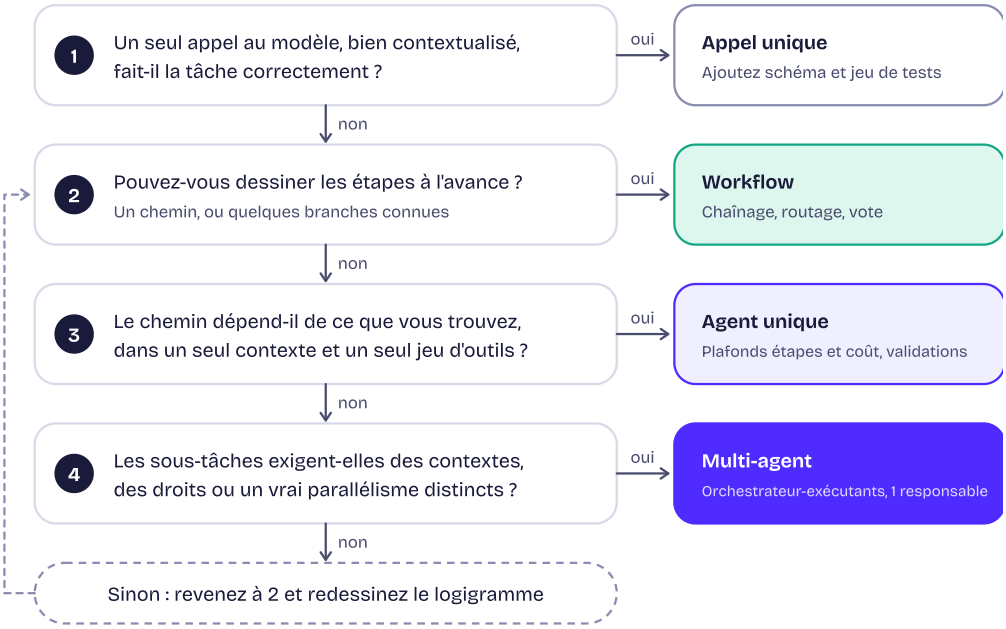
- **Le chaînage** n'est fiable que grâce aux points de contrôle : du code simple, entre les appels, vérifie le schéma, les champs obligatoires et la longueur.
- **Le routage** est votre principal levier de coût (petits modèles pour les cas faciles). Donnez toujours au routeur une catégorie « incertain » qui part vers un humain ou un chemin plus puissant ; un routeur obligé de choisir choisira avec assurance, et à tort.
- **Le vote** détecte l'incertitude à peu de frais : un désaccord entre exécutions est un signal.
- **L'évaluateur-optimiseur** a besoin d'une condition d'arrêt et d'un plafond d'itérations. Le critique trouve toujours quelque chose.
- **La boucle d'agent** (nommée d'après l'article ReAct) exige un *harness* (le code qui encadre la boucle du modèle : budget, garde-fous, état, télémétrie) complet. Arize décrit des agents qui interrogent sans cesse l'état d'une tâche au lieu d'attendre, transformant une tâche en centaines d'appels d'API alors que chaque ligne de log a l'air saine (Arize).
- **Planifier puis exécuter** donne aux humains un plan à relire avant que quoi que ce soit ne s'exécute : un point d'approbation naturel.

L'arbre de décision : appel unique, workflow, agent, multi-agent

Parcourez cet arbre pour chaque nouveau cas d'usage. Arrêtez-vous au premier « oui ».

Parcourez l'arbre, arrêtez-vous au premier oui

Chaque étape vers le bas coûte plus cher et se teste moins bien



> **Arrêtez-vous au premier oui.** Chaque boîte plus bas coûte plus cher et se teste plus difficilement, elle doit donc être justifiée par un cas d'évaluation en échec.

L'étape 4 porte sur les contextes, les permissions et le parallélisme, pas sur des « rôles » : découper ne paie pas parce que la tâche « ressemble » à une équipe. Par défaut, retenez la boîte la plus haute qui passe vos *evals* (évaluations automatisées : jeux de tâches avec résultat attendu et notation), et ne descendez qu'avec un cas d'évaluation en échec que de meilleurs prompts, contextes ou outils ne peuvent pas corriger. Notez ce cas : il justifie le surcoût.

Le budget de déterminisme : où mettre le code et où mettre le modèle

Chaque appel à un LLM ajoute de la variance, et des sorties fausses peuvent avoir l'air justes. Ne dépensez cette variance que là où le modèle apporte quelque chose que le code ne peut pas apporter.

> **Le modèle propose, le code dispose.**

Dans le code (réponse juste calculable)	Dans le modèle (langage et jugement)
Permissions et règles : qui voit quoi, quels remboursements exigent une approbation	De quoi parle ce message, à quel point semble-t-il urgent ?
Arithmétique, dates, échéances de SLA	Lequel de ces documents répond à la question ?
Recherches par clés connues : offre, contrat, responsable	Rédiger une réponse sur le bon ton
Routage sur des champs structurés, validation, nouvelles tentatives, idempotence	Résumer ce fil pour un humain

Le modèle propose, le code dispose. Le modèle renvoie une sortie structurée (une catégorie issue d'une énumération, un brouillon) ; le code la valide, applique les règles métier et décide. Un modèle qui annonce « priorité : P1 » pour un client de l'offre gratuite ne l'emporte pas sur la règle selon laquelle les offres gratuites n'ont pas de file P1. Le test : « si le modèle se trompe là-dessus, un test unitaire l'aurait-il détecté ? » Si oui, cela relève du code. « Les remboursements de plus de 500 euros exigent l'approbation d'un manager » est une règle, pas une ligne de prompt : les prompts ne font pas respecter les règles.

Exemple détaillé : le tri des tickets de support, construit de trois façons

La tâche. Un éditeur SaaS B2B reçoit des tickets de support par e-mail et par chat. Pour chacun, le système attribue une **catégorie** (facturation, technique, accès, fonctionnalité, autre), une **priorité** de P1 (panne pour un client payant) à P4 (question), stockée sous forme d'entier de 1 à 4, une **file** et, éventuellement, un **brouillon de réponse** tiré de la base de connaissances. Les P1 doivent atteindre l'astreinte en quelques minutes, les faux P1 réveillent des gens la nuit, et aucune réponse ne part sans un humain. À disposition : un CRM, une base de connaissances consultable, une page d'état des incidents ouverts.

Version A : le workflow

Cinq boîtes : classer, rechercher le client, appliquer les règles de priorité, router, rédiger. Deux ont besoin d'un modèle.

```
CATEGORIES = {"billing", "technical", "access", "feature", "other"}

def workflow_triage(ticket: Ticket) → Triage:
    # Appel au modèle 1 : petit modèle rapide, schéma JSON strict
    # (category, urgency_signal, confidence).
    c = classify(ticket.text, schema=ClassifySchema)
    if c.category not in CATEGORIES or c.confidence < 0.6:
        return to_human(ticket, reason="low_confidence")

    # Code : recherches par clé.
    customer = crm.get_customer(ticket.sender_email)
    if customer is None:
        return to_human(ticket, reason="unknown_sender")
    incident = status.open_incident_for(customer.region)

    # Code : les règles métier renvoient un entier,
    # de 1 (panne) à 4.
    inputs = PriorityInputs(
        category=c.category, urgency=c.urgency_signal,
        tier=customer.tier, open_incident=incident is not None,
    )
    priority = priority_rules(inputs)
    queue = QUEUE_MAP[(c.category, priority)]

    # Appel au modèle 2, seulement là où un brouillon aide.
    draft = None
    if c.category in {"billing", "access"}:
        docs = kb.search(ticket.text, top_k=4)
        draft = draft_reply(ticket.text, docs, customer.name)

    return Triage(c.category, priority, queue, draft,
                  confidence=c.confidence, inputs=inputs,
                  needs_review=True)
```

Deux appels au modèle au plus par ticket. La priorité vient des règles, donc un P1 signifie toujours la même chose, et chaque branche se teste avec des tests unitaires et des tickets passés étiquetés (100 à 200 répartis entre les catégories, avec une surreprésentation des cas rares et coûteux comme les vrais P1). Vous pouvez le chiffrer avant d'écrire le code : nombre de tickets × *tokens* (unités de texte facturées par le modèle) par ticket × prix.

D'où vient le score de confiance. Une confiance que le modèle écrit dans son propre JSON est un indice, pas une probabilité ; le 0,6 est illustratif. Regroupez les tickets étiquetés par confiance déclarée, mesurez l'exactitude de chaque groupe et placez le seuil là où l'exactitude atteint l'exigence métier, probablement par catégorie. Si les scores déclarés ne suivent pas l'exactitude, utilisez l'accord entre deux ou trois exécutions, ou les log-probabilités des tokens quand votre fournisseur les expose.

Ce qu'il ne sait pas faire : enquêter. « L'export est cassé depuis votre mise en production de mardi, voir le log joint » reçoit une catégorie et une priorité, mais personne ne vérifie si la mise en production de mardi a touché aux exports.

Version B : l'agent

Un seul modèle dans une boucle bornée, avec les outils `search_kb`, `get_customer`, `get_open_incidents`, `search_recent_tickets`, `read_attachment` et `submit_triage`. Le prompt contient les catégories, la politique de priorité et la table des files ; le harness plafonne l'exécution à 12 étapes et à un coût fixe ; tous les outils sauf `submit_triage` sont en lecture seule.

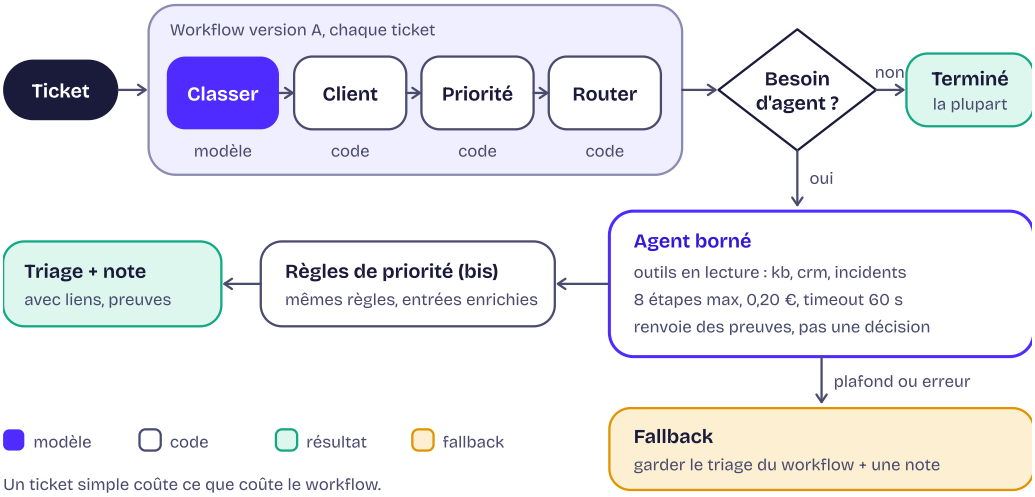
Sur le ticket de l'export de mardi, l'agent lit le log, trouve trois signalements récents similaires, remarque un incident ouvert et propose un P2 avec des liens : ce que ferait un ingénieur support confirmé. Le coût : 3 à 12 appels au modèle par ticket, chacun renvoyant un contexte qui grossit, avec une latence variable. La priorité est désormais la lecture par le modèle d'une politique rédigée en prose, donc des tickets similaires peuvent recevoir des priorités différentes, et les tests doivent couvrir les trajectoires. Sur un simple ticket « comment changer mon adresse de facturation ? », l'agent cherche et vérifie quand même. La plupart des tickets sont simples.

Version C : l'hybride

Faites passer chaque ticket par le workflow. N'envoyez à un agent borné que les tickets qui nécessitent une enquête, et laissez l'agent proposer, pas décider.

L'hybride : une ossature déterministe avec une section agentique bornée

L'agent propose des éléments de preuve. Le code applique les règles et a le dernier mot.



Un ticket simple coûte ce que coûte le workflow.

> **La colonne vertébrale d'abord, l'agent là où ça dépend.** Chaque ticket passe par le workflow ; seuls les tickets qui nécessitent une enquête atteignent un agent borné en lecture seule, et c'est toujours le code qui fixe la priorité.

```
from dataclasses import replace

def triage(ticket: Ticket) → Triage:
    # Version A, inchangée.
    base = workflow_triage(ticket)
    needs_investigation = base.category == "technical" and (
        base.priority ⩽ 2 or ticket.has_attachment
        or base.confidence < 0.8
    )
    # La plupart des tickets s'arrêtent ici.
    if not needs_investigation:
        return base

    result = run_agent(
        task=Task(id=ticket.id,
                  input=investigate_prompt(ticket, base)),
        model=SMALL_MODEL,
        # kb, crm, incidents, tickets
        tools=READ_ONLY_TOOLS, flags=FLAGS,
        budget=Budget(8, 0.20, 60),
    )
    # Un passage de relais garde le tri du workflow,
    # avec la raison.
    if result.status ≠ "done":
        return replace(base, note="enquête incomplète : "
                       + result.reason)
    # Un Finding validé : l'argument de submit_finding.
    finding = result.output

    # Ne garder que les preuves qui pointent vers de vrais
    # enregistrements (ticket, incident, ligne de log).
    evidence = verify_evidence(finding.evidence)

    # L'agent propose ; le code applique les mêmes règles
    # à des entrées plus riches.
    inputs = replace(base.inputs, evidence=evidence)
    p = priority_rules(inputs)
    return replace(base, priority=p, inputs=inputs,
                   queue=QUEUE_MAP[(base.category, p)],
                   note=finding.summary, links=finding.links)
```

`verify_evidence` protège contre la défaillance la plus pernicieuse : une preuve fausse mais plausible. L'agent doit étayer chaque affirmation par un identifiant d'enregistrement ; le code récupère chaque enregistrement et écarte les affirmations dont l'identifiant n'existe pas ou ne correspond pas (un « incident ouvert » qui est clos). Les intuitions non vérifiées peuvent figurer dans le résumé, mais ne font jamais passer un ticket en P1.

Les tickets simples coûtent ce que coûte le workflow, et un P1 signifie toujours ce que disent les règles. Si l'agent échoue ou épuise son budget, le ticket garde le tri du workflow avec une note : une dégradation maîtrisée, pas une panne.

Régler le déclencheur. `needs_investigation` décide à la fois du coût et de la couverture. Journalisez-le sur chaque ticket et suivez deux taux hebdomadaires : les **enquêtes manquées** (pas d'enquête, puis escalade par un humain : déclencheur trop strict) et les **enquêtes inutiles** (le résultat n'a rien changé : vous payez sans valeur). Modifiez une condition à la fois.

La comparaison

Ce sont des estimations de conception sans cache, pas des benchmarks. Le nombre d'appels sous-estime l'écart : une classification lit un ticket, alors que chaque étape d'enquête renvoie un contexte qui grossit. Avec des volumes de tokens illustratifs (environ 1 200 tokens d'entrée par classification et environ 2 800 par ticket pour la version A une fois les brouillons comptés ; une enquête en six étapes qui passe de 5 000 à 11 000 tokens, soit environ 48 000 au total), la version C coûte environ 1,85 fois la version A quand 5 % des tickets font l'objet d'une enquête, et environ 4,4 fois à 20 %. C'est le déclencheur, pas le modèle, qui fixe la facture.

L'exemple illustratif de remboursement d'ADK 2.0 publié par Google, sur des API simulées, va dans le même sens : des nœuds déterministes font passer les tokens par exécution de 5 152 à 2 265 et le temps de 7,2 à 5,7 secondes par rapport à un simple agent LLM (Google Developers Blog, juillet 2026, affirmation de l'éditeur).

Dimension	A. Workflow	B. Agent	C. Hybride
Appels au modèle par ticket	1 à 2, fixe	3 à 12, choisis par le modèle	1 à 2 pour la plupart ; jusqu'à 2 + 8 en cas d'enquête
Coût modèle relatif	1 fois	1,5 à 12 fois selon le nombre d'appels, davantage avec la croissance du contexte	Environ 1,85 fois à 5 % d'enquêtes, 4,4 fois à 20 % (illustratif)
Latence	Faible, prévisible	Variable, plafonnée	Faible pour la plupart des tickets
Cohérence des priorités	Déterministe	Varie entre tickets similaires	Déterministe, entrées plus riches
Tickets inédits, multi-systèmes	Un humain enquête	Oui	Oui, pour les tickets routés
Comportement en cas d'échec	File humaine	Atteint un plafond ; pas de tri sans *fallback* (solution de repli : autre modèle ou chemin si le premier échoue)	Repli sur le tri du workflow
Comment le tester	Tests unitaires plus jeu étiqueté	Evals de trajectoire et de résultat, relecture humaine	Tests du workflow plus un petit jeu de trajectoires

Le verdict

Par défaut, choisissez la version A : un système qui fonctionne, un jeu de données étiqueté et un coût de référence en quelques jours. Livrez la version C quand vous disposez d'assez de tickets mal traités par A faute d'enquête pour constituer un jeu d'évaluation. Ne livrez la version B que si la plupart des tickets exigent une enquête, que le volume est faible et que vous acceptez que les priorités deviennent des jugements.

L'hybride est la forme générale des bons systèmes d'agents : une colonne vertébrale déterministe pour le travail prévisible, des sections agentiques bornées là où la découverte paie, et le code qui a le dernier mot sur les décisions qui comptent. Le chapitre 17 emmène ce système en production.

Règle de décision

Règle de placement de l'hybride

Placez l'agent là où le logigramme dit « ça dépend », donnez-lui des outils en lecture seule et un petit plafond, faites-lui renvoyer des preuves plutôt que des décisions, et laissez le code appliquer les règles. Si sa sortie doit déclencher une action directe (une écriture dans une fiche client), placez une approbation humaine ou une validation déterministe entre l'agent et l'action.

Anti-patterns

- **L'agent pour une tâche cron.** Un rapport nocturne construit comme un agent redécouvre le plan chaque nuit, saute un indicateur jugé « pas pertinent aujourd'hui », et les rapports cessent d'être comparables. Récupérez et comparez en code, puis un seul appel au modèle rédige le résumé.
- **Le théâtre multi-agent.** Un « agent PDG » délègue à des agents « manager », « analyste » et « QA ». Chaque passage de relais perd du contexte, les coûts se multiplient à chaque saut, et personne ne sait dire quel agent a causé une sortie fausse. C'est généralement une chaîne de prompts déguisée : appliquez l'étape 4 de l'arbre.
- **L'agent comme gestionnaire d'erreurs.** Envelopper une intégration fragile dans un agent « pour qu'il comprenne ce qui n'a pas marché ». Corrigez l'intégration : les agents lisent parfois une erreur HTTP 500 sans texte d'échec explicite comme « traité » (Arize).
- **Le même gros modèle partout.** La classification et l'extraction tournent généralement très bien sur un petit modèle. Routez par tâche et évaluez chaque route.

Notes par parcours

- **Tech.** Construisez d'abord la version A, même si vous êtes sûr d'avoir besoin d'un agent. Gardez typée l'interface entre workflow et agent, avec les trois statuts du chapitre 8 : terminé, passage de relais, en attente d'un humain.

- **Manager.** Utilisez l'arbre de décision comme point de contrôle en revue : chaque proposition d'agent nomme l'étape qu'elle ne franchit pas, avec un exemple. Demandez le coût par tâche réussie et ce qui se passe au plafond.
- **Solo.** Les workflows échouent bruyamment et de façon prévisible, vous passerez donc moins de soirées à déboguer. Notez dans un tableur le coût de chaque étape plus votre temps de relecture : cette somme, calculée pour le cas avec nouvelle tentative, est votre prix plancher (chapitre 13).
- **No-code.** Les flux n8n, Zapier ou Make sont déjà des workflows : le routage est un aiguillage après une étape de classification, le chaînage est une suite d'étapes LLM avec des filtres, un nœud IF sur le champ de confiance envoie les scores faibles vers une file humaine, et l'enquête est un nœud AI agent placé derrière une condition, avec des outils en lecture seule, une limite d'itérations basse et une étape d'approbation avant tout message au client.

Checklist

- [ ] Nous avons dessiné le logigramme à partir d'au moins cinq entrées réelles avant de choisir une architecture.
- [ ] Nous avons noté à quelle étape de l'arbre de décision notre conception échoue, avec un exemple.
- [ ] Chaque décision dont la réponse est calculable est dans le code, pas dans un prompt.
- [ ] Chaque sortie du modèle est validée, et chaque classifieur a un chemin « incertain ».
- [ ] Chaque section agentique a des plafonds, un *timeout* (délai d'expiration) et des outils en lecture seule sauf écritures approuvées, et se replie sur un résultat de workflow ou sur un humain.
- [ ] Nous avons une référence de coût et de qualité issue d'un workflow, à laquelle comparer toute conception agentique.

Le chapitre 2 en une page

Les workflows avant les agents

Si vous pouvez dessiner le logigramme, construisez le workflow. N'utilisez un agent que pour les boîtes marquées « ça dépend ».

À retenir

- Parcourez l'arbre de décision depuis le haut et arrêtez-vous au premier oui
- Le modèle propose, le code dispose
- Livrez d'abord le workflow ; passez à l'hybride quand de vrais tickets en montrent le besoin

À faire dès lundi

- Passez une heure avec la personne qui fait la tâche aujourd'hui et notez chaque « ça dépend »
- Cherchez dans vos prompts les seuils, les montants et les « doit » : chacun est une règle à déplacer dans le code
- Chiffrez la version workflow : nombre de tickets × tokens par ticket × prix

1 à 2
appels au modèle par ticket, tri par workflow (estimation de conception)

3 à 12
appels au modèle par ticket, tri par agent (estimation de conception)

Pour aller plus loin : le livre complet

Ceci était un extrait gratuit d'**Agents en production**, un guide pratique pour faire passer des agents du pilote à la production et les y maintenir. Le livre complet comprend un prologue, 21 chapitres en neuf parties, une postface sur la façon dont le livre a été fait et la checklist de mise en production, plus un cahier d'exercices compagnon et des compléments gratuits en ligne.

Le livre complet couvre :

- **Modèles mentaux :** ce qu'est vraiment un agent, et quand un simple *workflow* (enchaînement d'étapes prédéfinies dans le code) vaut mieux.
- **Briques de base :** des outils que les agents peuvent vraiment utiliser, MCP après la révision 2026-07-28, les *skills* (dossiers avec un `SKILL.md`, chargés à la demande), les sous-agents et les plugins, l'ingénierie du contexte et la mémoire.
- **Frameworks et architecture :** une carte indépendante des éditeurs du paysage des frameworks en 2026, un guide de décision et des architectures de référence pour les agents pilotés par les événements et durables.
- **Production :** des *harnesses* (le code qui encadre la boucle du modèle) de fiabilité, des *evals* (évaluations automatisées) en CI, l'observabilité et l'économie unitaire, la sécurité des agents et de MCP, la gouvernance et le règlement européen sur l'IA.
- **Business :** des recettes d'automatisation no-code, des guides pratiques pour indépendants qui font gagner ou économiser de l'argent, et un plan construire, acheter et recruter pour les managers.
- **L'usine logicielle agentique :** des agents qui mènent le logiciel de l'idée à la *pull request* (demande de fusion) fusionnée, le testent de bout en bout et continuent de l'améliorer, avec un projet de synthèse qui fait passer un système par chaque poste.
- **La suite :** un atlas des cas d'usage, les preuves sur la productivité, l'emploi, le coût et l'énergie, et des prévisions sourcées.
- **Annexe :** la checklist de mise en production (59 contrôles, dont 19 obligatoires). En ligne et gratuits : une checklist de conception des outils et serveurs MCP, un modèle de modélisation des menaces, les notes de terrain 2026, un glossaire et les consignes de chaque agent qui a écrit le livre. Le cahier d'exercices compagnon contient les exercices et leurs corrigés.

Quatre parcours de lecture (Tech, Manager, Solo, No-code) vous disent exactement quels chapitres lire et lesquels sauter. Les exercices, le projet fil rouge et les autoévaluations sont fournis avec leurs corrigés complets.

Écrit par des agents IA (Claude, un modèle conçu par Anthropic) sous la direction de Yann.

Disponible sur agentinprod.com/fr/