Notilis Tech

Orchestrer à l'échelle d'une équipe : subagents, Symphony et gouvernance technique

Penser l'orchestration d'équipe avec subagents, état, règles, espaces de travail, observabilité et gouvernance technique.

2026-06-07 · Subagents · Orchestration · Gouvernance

Résumé

Passer à l'échelle ne consiste pas à lancer plus d'agents plus vite. Plus il y a d'agents, plus les entrées, sorties, validations, permissions et traces doivent être explicites.

L'orchestration arrive après la discipline : harness de projet, task briefs vérifiables, boucle de validation, workflows réutilisables, puis skills, subagents et automations.

Cet article montre comment organiser cette progression avec des subagents, Symphony comme modèle d'orchestration, et une gouvernance technique adaptée à une équipe. Je parle surtout de Codex parce que c'est mon outil quotidien, mais les principes valent aussi pour les autres agents de codage et leurs mécanismes équivalents.

Sommaire

Contexte de lecture

Audience

Cet article s’adresse aux tech leads, platform engineers et équipes qui veulent coordonner plusieurs agents ou workflows d’agents de codage à l’échelle d’un collectif.

Ce que cet article couvre

Il présente la délégation progressive, les subagents, Symphony comme spécification d’orchestration, deux cas pratiques et les dimensions de gouvernance technique.

Ce que cet article ne couvre pas

Il ne vend pas l’orchestration comme un raccourci magique et ne propose pas de déléguer les décisions critiques sans responsabilité humaine.

Plus d'agents exige plus de structure

Un agent seul peut déjà échouer si la tâche est floue. Plusieurs agents peuvent échouer plus vite, en parallèle, avec des résultats incohérents. Le problème n'est pas la parallélisation en elle-même. Le problème est la parallélisation d'un travail qui n'est pas encore découpé.

Avant de déléguer en parallèle, il faut savoir :

```
Quel problème est divisible ?
Quelles sorties chaque sous-tâche doit-elle produire ?
Quels fichiers ou domaines chaque agent peut-il inspecter ?
Quelles validations s'appliquent à chaque résultat ?
Qui consolide les résultats ?
Quelles contradictions doivent être arbitrées par un humain ?
```

Sans ces réponses, les subagents produisent souvent des analyses utiles mais difficiles à assembler, ou des implémentations qui se chevauchent.

L'orchestration n'est donc pas un raccourci pour éviter le cadrage des tâches. C'est une couche supplémentaire qui exige des tâches encore mieux définies.

Orchestration progressive

Une équipe peut organiser sa montée en orchestration par paliers.

- Le premier palier est l'usage individuel interactif. Un développeur travaille avec un agent de codage sur une tâche bornée, valide, relit et produit une evidence note.
- Le deuxième palier est le workflow réutilisable. La tâche revient souvent ; elle devient une procédure Markdown que l'équipe peut rejouer.
- Le troisième palier est la skill. Le workflow est suffisamment stable pour être encapsulé sous forme de capacité réutilisable, avec des entrées, des sorties et des critères de validation attendus.
- Le quatrième palier est la délégation parallèle ponctuelle. Dans Codex, cela peut passer par des subagents ; dans d'autres outils, par des agents spécialisés, des workers ou des runs parallèles qui explorent différents aspects d'un problème : sécurité, tests, bugs, maintenabilité, performance ou analyse de dépendances.
- Le cinquième palier est l'automation contrôlée. Certains workflows récurrents peuvent être planifiés, surveillés et relus.
- Le sixième palier est l'orchestration d'équipe. Les tâches, espaces de travail, statuts, validations et reviews sont gérés comme un système de production, pas comme une collection de chats.

La progression peut être résumée ainsi :

Palier	Usage	Condition de maturité
1	Agent interactif	Task brief clair
2	Workflow Markdown	Procédure répétable
3	Skill	Entrées/sorties stables

Palier	Usage	Condition de maturité
4	Subagents	Travail divisible et synthèse claire
5	Automation	Workflow stable et faible ambiguïté
6	Orchestration d'équipe	État, règles, espaces de travail et observabilité

Le point essentiel est que chaque palier dépend du précédent. Une équipe qui saute directement à l'orchestration risque surtout d'automatiser ses ambiguïtés.

Quelques définitions évitent de mélanger les niveaux.

Terme	Rôle
Workflow Markdown	Procédure écrite, relue par l'équipe, que l'on peut donner à l'agent pour répéter une tâche.
Skill	Capacité réutilisable, plus stable qu'un simple workflow, avec des entrées, sorties et validations attendues.
Subagent	Agent lancé en parallèle dans un run pour traiter un axe ou une sous-tâche.
Automation	Workflow déclenché de manière récurrente ou événementielle, avec supervision et critères d'arrêt.
Orchestration d'équipe	Système qui coordonne plusieurs tâches, agents, espaces de travail, statuts, validations, reviews et escalades.

La distinction est importante. Une skill n'est pas une automation. Un subagent n'est pas un membre permanent de l'équipe. Une automation n'est pas une autorisation de modifier le produit sans contrôle. Et une orchestration d'équipe n'est pas un gros prompt : c'est un système de travail.

Subagents : utiles pour explorer, dangereux pour compenser un mauvais scope

OpenAI documente les subagents comme des workflows où Codex peut lancer des agents spécialisés en parallèle, puis collecter leurs résultats dans une réponse consolidée. La documentation précise aussi que Codex ne lance des subagents que lorsqu'on le demande explicitement, et que ces workflows consomment davantage de tokens qu'un run mono-agent comparable.[1]

Les subagents sont particulièrement utiles lorsque le travail est naturellement parallèle. Par exemple, une review de PR peut être découpée en axes : sécurité, bugs, tests, maintenabilité, conditions de concurrence, performance. Une exploration de base de code peut être découpée par modules. Une migration peut être analysée par service.

Une bonne demande de subagents ressemble à ceci :

```
Relis la branche courante par rapport à main.

Lance un subagent par axe de review :
1. Sécurité et secrets
2. Risque de régression
3. Tests manquants
4. Compatibilité des contrats API
5. Maintenabilité

Chaque subagent doit retourner :
- constats
- preuves issues des fichiers
- sévérité
- action recommandée
- niveau de confiance

Attends tous les subagents, puis produis une review consolidée avec conflits et priorités.
```

Ce prompt ne demande pas seulement « lance plusieurs agents ». Il définit les axes, les sorties et la consolidation.

Les subagents sont moins adaptés aux tâches où le scope est flou, où les agents risquent de modifier les mêmes fichiers, ou où la décision dépend fortement d'un jugement humain non formalisé. Dans ces cas, il vaut mieux commencer par une phase de reconnaissance ou un plan.

Approbations, sandbox et subagents

Les subagents héritent du contexte opérationnel. La documentation OpenAI indique notamment que les subagents héritent des règles de sandbox courantes, et que les demandes d'approbation peuvent remonter depuis des threads inactifs dans les sessions CLI interactives.[2]

Cela signifie qu'une équipe doit penser les subagents comme des agents réels, pas comme de simples prompts secondaires. Ils peuvent utiliser des outils, consommer des tokens, demander des approbations, et produire des résultats qui doivent être consolidés.

Avant de lancer des subagents, il faut donc vérifier :

```
La sandbox est-elle adaptée ?
Les agents doivent-ils lire seulement ou modifier ?
Les tâches parallèles touchent-elles les mêmes fichiers ?
Qui arbitre les conflits ?
Le résultat attendu est-il une analyse, un plan ou un patch ?
```

Pour la plupart des équipes, le premier usage raisonnable des subagents est l'analyse parallèle, pas l'implémentation parallèle. Une analyse parallèle produit des signaux. Une implémentation parallèle produit des diffs qui peuvent entrer en conflit.

Cas pratique 1 : review de PR par axes

Le cas le plus simple pour introduire les subagents est la review de PR. Le travail est naturellement divisible, le risque est limité si l'on reste en lecture seule, et le résultat peut être consolidé par un humain.

Le contexte : une branche contient une évolution fonctionnelle. Le développeur veut une review assistée avant de demander la review humaine. L'objectif n'est pas de remplacer le reviewer. L'objectif est de produire

un dossier de review plus propre : risques, zones modifiées, tests manquants, contrats à vérifier.

Comment le faire

Avant de lancer l'agent, l'équipe doit avoir trois éléments : la branche à relire, la branche de référence, et les commandes de validation attendues. Par exemple :

```
Branche de référence : main
Branche courante : feature/invoice-export
Commandes de validation :
- npm run lint
- npm test
- npm run build
```

Ensuite, on lance un run explicitement orienté analyse. La consigne doit interdire les modifications de fichiers si l'on veut éviter que les subagents produisent des patches divergents.

```
Tu relis la branche courante par rapport à main.

**NE PAS MODIFIER LES FICHIERS. CECI EST UN RUN D'ANALYSE UNIQUEMENT.**

Lance un subagent par axe :
1. Sécurité et secrets
2. Risque de régression
3. Tests manquants ou faibles
4. Compatibilité des contrats API
5. Maintenabilité et architecture

Chaque subagent doit inspecter le diff et les fichiers proches pertinents.
Chaque subagent doit retourner ses constats avec ce format :

- Titre
- Severity: critical | high | medium | low | informational
- Evidence: chemin de fichier et symbole ou plage de lignes pertinente quand disponible
- Explanation
- Action recommandée
- Confidence: high | medium | low

Après la fin de tous les subagents, consolide les résultats en :
1. Synthèse de review
2. Constats bloquants
3. Constats non bloquants
4. Commandes de validation suggérées
5. Questions pour le reviewer humain
6. Zones sans problème significatif détecté
```

La sortie attendue n'est pas « la PR est bonne » ou « la PR est mauvaise ». Une bonne sortie ressemble plutôt à un dossier de review :

```
Synthèse de review
La branche est globalement cohérente, mais elle change le contrat d'export des factures et
manque de couverture de régression pour les lignes de facture vides.

Constats bloquants
- Test de régression manquant pour les lignes de facture vides
  Severity: high
  Evidence: src/invoices/export.service.ts, tests/invoices/export.spec.ts
  Action recommandée : ajouter un test couvrant les lignes vides et les codes de taxe nuls
  Confidence: high

Constats non bloquants
- Le message d'erreur diffère des conventions API existantes
  Severity: low
  Evidence: src/api/invoices.controller.ts
  Action recommandée : aligner avec le format d'erreur existant
  Confidence: medium

Questions pour le reviewer humain
- L'ordre des colonnes CSV fait-il partie du contrat public ?
- Les exports doivent-ils être stables entre locales ?
```

La décision reste humaine. Si le résultat est exploitable, le reviewer peut demander un second run plus ciblé : « propose a minimal patch for the missing tests only ». C'est cette séparation qui rend le workflow sain : d'abord l'analyse parallèle, ensuite seulement l'implémentation bornée.

Cas pratique 2 : préparer un refactoring large sans lancer cinq agents dans les mêmes fichiers

Le deuxième cas utile est la préparation d'un refactoring ou d'une migration. C'est aussi le cas où il faut être prudent. Si l'on demande à plusieurs agents de modifier directement les mêmes zones, on obtient vite des diffs contradictoires. Le bon usage des subagents consiste donc à cartographier le travail avant de le découper.

Le contexte : une équipe veut remplacer un ancien client HTTP interne par une nouvelle abstraction. Le code est réparti dans plusieurs modules. Le risque n'est pas seulement technique : il faut identifier les contrats implicites, les tests existants, les zones non couvertes et les lots de migration.

Comment le faire

La première passe doit être une passe d'analyse. Elle doit produire une carte de migration, pas un patch.

```
Nous devons migrer de LegacyHttpClient vers HttpGateway.

**NE PAS MODIFIER LES FICHIERS. CECI EST UN RUN D'ANALYSE UNIQUEMENT.**

Lance des subagents par zone d'investigation :
1. Inventaire des usages : trouver tous les usages de LegacyHttpClient et les grouper par
module
2. Analyse des contrats : identifier les hypothèses implicites de requête/réponse
3. Couverture de tests : identifier les tests existants et la couverture manquante
4. Analyse des risques : identifier les risques runtime, sécurité, retry, timeout et gestion
d'erreur
5. Découpage de migration : proposer des lots d'implémentation sûrs qui évitent les
chevauchements de propriété de fichiers

Chaque subagent doit retourner :
- Scope inspecté
- Constats avec preuves issues des fichiers
- Risques
- Limites de lots proposées
- Fichiers qui ne doivent pas être modifiés ensemble par différents agents
- Niveau de confiance

Après la fin de tous les subagents, produis un plan de migration avec :
1. Carte des dépendances
2. Lots recommandés
3. Commandes de validation par lot
4. Fichiers nécessitant une review humaine
5. Critères d'arrêt
6. Prompt du premier lot utilisable par l'agent
```

La sortie intéressante est une carte de travail. Par exemple :

```
Lots de migration recommandés

Lot 1 : adapter partagé et tests
Files:
- src/http/http-gateway.ts
- src/http/http-gateway.spec.ts
Validation:
- npm test -- http-gateway
Critères d'arrêt :
- arrêter si la sémantique publique de retry diffère de LegacyHttpClient

Lot 2 : migration du module Invoice
Files:
- src/invoices/invoice-api.ts
- src/invoices/invoice-api.spec.ts
Validation:
- npm test -- invoices
Review humaine :
- confirmer que les codes d'erreur exportés restent stables

Lot 3 : migration du module Shipment
Files:
- src/shipments/shipment-api.ts
- src/shipments/shipment-api.spec.ts
Validation:
- npm test -- shipments
Review humaine :
- vérifier le comportement de timeout en mode hors ligne
```

À partir de là, l'équipe peut lancer l'agent sur un seul lot à la fois. Les subagents ont servi à découper, pas à produire directement cinq migrations concurrentes. C'est souvent la différence entre une orchestration utile et une orchestration qui crée du travail de nettoyage.

Symphony : lire une spécification d'orchestration, pas une promesse magique

Symphony est un bon exemple parce qu'il montre une manière de penser l'orchestration comme système. OpenAI le présente comme une spécification écrite fonctionnant comme un superviseur pour orchestrer le travail agentique, avec un gestionnaire d'issues servant de plan de contrôle.[3] Le repository **openai/symphony** publie cette spécification et une implémentation de référence expérimentale.[4] L'article OpenAI précise aussi que Symphony est pensé comme une couche d'orchestration volontairement minimale et comme une référence, pas comme un produit autonome maintenu pour tous les cas d'usage.[5]

Je ne conseille donc pas de l'utiliser tel quel. C'est plus un lab à mettre en place pendant quelques jours ou semaines pour se poser les bonnes questions d'orchestration, pas un framework à adopter tel quel.

Le point intéressant n'est donc pas de dire « il faut utiliser Symphony ». Le point intéressant est de lire ce que Symphony clarifie : espace de travail, état, statut de tâche, règles, observabilité, redémarrage, passage de relais, cycle de vie de PR, CI, tentatives, retours et responsabilité.

Dans un workflow interactif, beaucoup de ces choses restent implicites. Le développeur sait quelle tâche est en cours, quel checkout est utilisé, quelles commandes ont été lancées, quel état a échoué, quelle PR attendre. Dans un système orchestré, ces informations doivent être explicites.

C'est la vraie leçon : à l'échelle d'une équipe, un agent ne travaille plus seulement dans un chat. Il travaille dans un système d'état.

Ce que Symphony rend visible	Ce qu'il faut en retenir	Ce qu'il ne faut pas en conclure
Gestionnaire d'issues comme plan de contrôle	Les tâches doivent avoir un statut, un owner et une trace	Un gestionnaire d'issues suffit à faire une orchestration mature
Espace de travail isolé	Chaque agent doit travailler dans un espace contrôlé	L'isolation supprime le besoin de review
Règles et permissions	Les capacités de l'agent doivent être explicites	Plus de permissions donne mécaniquement de meilleurs résultats
Cycle de vie de PR et CI	Le travail agentique doit rejoindre le cycle de livraison existant	L'agent peut décider seul du merge
Retours et tentatives	Les erreurs doivent réintégrer le système	Il faut relancer indéfiniment jusqu'à obtenir un résultat

Symphony vaut donc surtout comme modèle mental. Il force à se poser les questions que l'on évite souvent quand on travaille seulement dans une conversation interactive.

Les dimensions d'une spécification d'orchestration

Une spécification d'orchestration devrait répondre à plusieurs questions.

- La première est l'espace de travail. Où l'agent travaille-t-il ? Dans un worktree ? Dans un environnement local ? Dans un container ? Dans un espace de travail distant ? Comment l'isolation est-elle garantie ? Comment les dépendances sont-elles installées ? Comment les conflits sont-ils gérés ?
- La deuxième est l'état. Quel est le statut d'une tâche ? Que signifient prêt, en cours, bloqué, en attente de review, en test, en merge ou terminé ? Qui peut changer le statut ? Quel événement déclenche le prochain état ?
- La troisième est la règle d'exécution. Quels fichiers peuvent être modifiés ? Quelles commandes sont autorisées ? Quels secrets sont interdits ? Quels niveaux de permission sont acceptés ? Quelles tâches demandent une review senior ?
- La quatrième est l'observabilité. Quels logs sont conservés ? Quelles commandes ont été lancées ? Quels tests ont échoué ? Quelle evidence note est produite ? Quels coûts ou tokens sont suivis ? Où les erreurs sont-elles visibles ?
- La cinquième est la review. Qui relit quoi ? Comment les commentaires de PR sont-ils traités ? Quand l'agent peut-il corriger ? Quand doit-il demander arbitrage ?
- La sixième est l'arrêt. Quand une tâche doit-elle être stoppée ? Quand un agent doit-il être redémarré ? Quand faut-il escalader à un humain ? Quand faut-il abandonner une piste ?

Sans ces dimensions, l'orchestration reste un ensemble d'agents lancés en parallèle. Avec ces dimensions, elle devient un système de livraison assistée.

Gouvernance technique : mettre les agents dans une boucle d'équipe responsable

À mesure que les agents interviennent dans plus de workflows, la gouvernance ne peut plus être traitée comme une réflexion tardive. Elle doit être intégrée au harness.

Pour une équipe de développement, la gouvernance technique couvre au minimum :

Dimension	Décision à expliciter
Permissions	Quels modes de sandbox et approbations selon les tâches ?
Scope	Quels fichiers, modules ou repositories peuvent être modifiés ?
Données	Quels secrets, PII ou données client sont interdits ?
Outils	Quels serveurs MCP, plugins ou intégrations sont autorisés ?
Validation	Quels checks sont obligatoires selon la criticité ?
Review	Qui relit les changements assistés par IA ?
Traçabilité	Où stocker evidence notes, logs et décisions ?
Escalade	Quand l'agent doit arrêter et demander un humain ?
Coût	Quels workflows consomment beaucoup et pourquoi ?

Dimension	Décision à expliciter
Critères d'arrêt	Quand arrêter une automation ou un pattern inefficace ?

Cette gouvernance ne doit pas devenir un mur administratif. Elle doit être proportionnée au risque. Mais elle doit exister, sinon l'équipe ne saura plus distinguer les usages maîtrisés des usages opportunistes.

MCP, plugins et outils externes à l'échelle d'équipe

Lorsqu'une équipe connecte un agent de codage à des outils externes via MCP, plugins, connecteurs ou intégrations équivalentes, elle augmente la surface d'action de l'agent. Cela peut être très utile : documentation à jour, design Figma, navigateur, logs, Sentry, GitHub, Linear ou autres systèmes. Mais cette extension doit être gouvernée.

Chaque outil externe ajoute trois questions : quelles données l'agent peut-il lire, quelles actions peut-il effectuer, et quelles traces sont produites ?

Un serveur MCP de documentation n'a pas le même niveau de risque qu'un outil capable d'agir sur un système de production. Un outil de lecture ne pose pas les mêmes problèmes qu'un outil de modification. Un outil utilisé en interactif n'a pas le même risque qu'un outil utilisé en automation.

La règle pratique :

```
Toute capacité ajoutée à l'agent doit avoir une règle, une validation et un owner.
```

Sinon, l'équipe augmente la puissance sans augmenter le contrôle.

Carte de délégation

Une carte de délégation peut aider à décider ce qui doit rester humain, ce qui peut être assisté, et ce qui peut être délégué.

Travail	Agent interactif	Subagents	Automation	Humain
Comprendre une zone de code	Oui	Oui si large	Rarement	Oriente le contexte
Corriger un bug local	Oui	Rarement	Si workflow stable	Review et merge
Review PR	Oui	Oui par axes	Possible sous règles	Décision finale
Refactoring local	Oui	Par analyse	Rarement	Contrats et architecture
Migration large	Par lots	Oui pour exploration	Non au début	Découpage et arbitrage
Sécurité critique	Assistance	Analyse complémentaire	Non sans gouvernance	Responsabilité principale

Travail	Agent interactif	Subagents	Automation	Humain
Documentation technique	Oui	Oui si large	Possible	Validation finale
Triage de CI instable	Oui	Oui	Possible	Priorisation

Cette carte n'est pas universelle. Elle doit être adaptée à l'équipe, au produit et au niveau de test. Mais elle empêche de déléguer au hasard.

Matrice de décision opérationnelle

La carte de délégation dit quel type de travail peut être confié à un agent, à des subagents ou à une automation. La matrice suivante aide à prendre les décisions opérationnelles au moment de lancer un run, d'ajouter un outil ou de monter le niveau d'autonomie.

Question développeur	Mauvaise réponse	Réponse actionnable
Est-ce que je peux lancer plusieurs agents sur ce sujet ?	Oui, plus il y en a, mieux c'est.	Oui seulement si le travail est divisible, si les sorties sont comparables et si la consolidation est prévue.
Est-ce que les subagents doivent modifier le code ?	Oui, chacun corrige sa partie.	Pas au début. D'abord analyse parallèle, puis patch borné sur un lot clair.
Est-ce qu'une skill remplace une automation ?	Oui, c'est la même chose.	Non. Une skill est une capacité réutilisable ; une automation est un déclenchement contrôlé dans le temps ou par événement.
Est-ce qu'un serveur MCP peut être ajouté librement ?	Oui, si l'outil est utile.	Non. Chaque outil doit avoir une règle, un owner, un périmètre de lecture/action et une trace.
Est-ce que Symphony doit être copié tel quel ?	Oui, puisque c'est une référence OpenAI.	Non. Il faut surtout reprendre la discipline : état, espace de travail, règles, observabilité, review et cycle de vie.
Quand arrêter un run ?	Quand l'agent dit qu'il a fini.	Quand les validations passent, que l'evidence note est exploitable, ou quand un critère d'arrêt impose l'escalade.
Qui prend la décision finale ?	L'agent, s'il a produit une bonne synthèse.	Le responsable humain désigné : reviewer, tech lead, maintainer ou owner fonctionnel selon le risque.

Roadmap de mise à l'échelle

Une feuille de route réaliste peut suivre quatre étapes.

Étape 1 : stabiliser le harness

Avant l'orchestration, l'équipe doit avoir **AGENTS.md**, commandes de validation, templates de task brief, matrice de review et evidence notes. Sans cela, chaque agent travaille avec trop d'implicite.

Étape 2 : standardiser les workflows

Les workflows récurrents sont documentés en Markdown : bugfix, review PR, refactoring local, documentation, tests, lot de migration. Chaque workflow décrit entrées, étapes, validation et format de sortie.

Étape 3 : introduire skills et subagents

Les workflows stables deviennent des skills. Les tâches naturellement parallèles peuvent utiliser des subagents pour l'exploration ou la review par axes. Les sorties doivent être standardisées pour faciliter la consolidation.

Étape 4 : automatiser et orchestrer sous gouvernance

Les automations ne concernent que les workflows stables. L'orchestration d'équipe exige un modèle d'état, d'espace de travail, de règles, d'observabilité et de review. Les tâches critiques restent sous supervision renforcée.

Cette feuille de route évite de construire trop tôt un système complexe. Elle laisse l'équipe apprendre sur des workflows simples avant d'augmenter le niveau d'autonomie.

Les erreurs fréquentes

- La première erreur consiste à croire que subagents signifie productivité automatique. Les subagents sont utiles lorsque le travail est divisible et les sorties comparables.
- La deuxième erreur consiste à paralléliser l'implémentation trop tôt. L'exploration parallèle est souvent plus sûre que le patch parallèle.
- La troisième erreur consiste à ignorer le coût de consolidation. Plusieurs agents produisent plusieurs résultats ; quelqu'un doit synthétiser, arbitrer et décider.
- La quatrième erreur consiste à traiter Symphony comme une solution clé en main. L'intérêt principal est la discipline d'orchestration qu'il illustre : gestionnaire d'issues, état, espace de travail, règles, observabilité et review.
- La cinquième erreur consiste à connecter des outils sans règle. Plus l'agent peut accéder à des systèmes, plus le contrôle doit être clair.

Checklist opérationnelle

Avant de passer à l'orchestration, il faut vérifier que

- [ ] les workflows simples fonctionnent
- [ ] les tâches sont bornées
- [ ] les validations sont exécutables
- [ ] les sorties sont standardisées
- [ ] les permissions sont définies
- [ ] les evidence notes existent
- [ ] les responsabilités humaines sont claires
- [ ] les conditions d'arrêt sont écrites.

Pour les subagents, il faut préciser

- [ ] les axes
- [ ] les sorties attendues
- [ ] la stratégie de consolidation
- [ ] le niveau de permission.

Pour les automations, il faut ajouter

- [ ] un propriétaire
- [ ] une cadence
- [ ] une condition d'arrêt
- [ ] une sandbox adaptée
- [ ] une review des premiers runs.

Conclusion

L'orchestration d'agents de codage n'est pas une étape magique. C'est une extension du harness de projet.

Un agent seul exige déjà du contexte, du scope, de la validation et une review. Plusieurs agents exigent en plus des contrats d'entrées/sorties, une consolidation, un modèle de règles, un modèle d'espace de travail, un modèle d'état et de l'observabilité. Symphony est intéressant parce qu'il rend ces dimensions visibles sous forme de spécification d'orchestration, pas parce qu'il supprimerait le besoin de discipline.

La bonne trajectoire est donc progressive : harness, tâches vérifiables, boucle qualité, workflows, skills, subagents, automations, orchestration. Plus l'équipe monte dans cette trajectoire, plus le rôle du développeur devient architectural : définir le système de travail dans lequel les agents peuvent produire sans diluer la responsabilité humaine.

Articles liés

- Un agent de codage n'est pas un chatbot

- Industrialiser les workflows : Markdown avant frameworks, skills avant automations
- Fermer la boucle qualité : tests, review humaine et evidence notes

Références

1. OpenAI Developers, « Subagents — Codex », consulté le 2026-06-07.
https://developers.openai.com/codex/subagents

2. OpenAI Developers, « Subagents — Codex », sections sur approbations et contrôles de sandbox, consulté le 2026-06-07.
https://developers.openai.com/codex/subagents

3. OpenAI, « An open-source spec for Codex orchestration: Symphony », consulté le 2026-06-07.
https://openai.com/index/open-source-codex-orchestration-symphony/

4. OpenAI, « openai/symphony », repository GitHub, consulté le 2026-06-07.
https://github.com/openai/symphony

5. OpenAI, « An open-source spec for Codex orchestration: Symphony », note sur Symphony comme couche minimale et référence, consulté le 2026-06-07.
https://openai.com/index/open-source-codex-orchestration-symphony/