GIT À L'ÈRE DES AGENTS IA

Le playbook opérationnel pour coordonner ingénieurs, agents et sous-agents sans perdre le contrôle de l'application

Victor PEREZ

EXTRAIT GRATUIT — CHAPITRES 1 ET 2

Ingénierie multi-agent · Git · Intégration · Récupération

Avant-propos

Le code est désormais produit plus vite qu'une équipe humaine ne peut le comprendre, le revoir et l'intégrer.

Ce livre s'adresse aux ingénieurs, tech leads, staff engineers et responsables qui utilisent des agents et sous-agents sur une même application. Il ne cherche pas à réenseigner Git. Il construit les solutions opérationnelles devenues nécessaires lorsque la production de code change d'échelle.

Les pratiques proposées sont indépendantes d'un fournisseur. Elles peuvent être appliquées avec Codex, GitHub Copilot, d'autres agents ou une infrastructure interne. Les produits évoluent ; les invariants restent : base identifiable, espace isolé, périmètre explicite, preuves reproductibles, intégration contrôlée et responsabilité humaine.

> **MODE D'EMPLOI**
>
> **Chaque chapitre produit une décision applicable**
>
> Vous trouverez des contrats YAML, politiques, prompts, commandes, checklists et modèles de tableaux de bord. Adaptez les seuils à votre équipe, mais ne supprimez pas la fonction de contrôle qu'ils remplissent.

Table des matières

PARTIE I

CE QUE LES AGENTS CHANGENT DANS L'USAGE DE GIT

CHAPITRE 1 — QUAND UN DÉVELOPPEUR DEVIENT RESPONSABLE DE PLUSIEURS PRODUCTEURS DE CODE

Le changement d'échelle que Git ne peut pas absorber seul

Le problème concret

Un développeur travaille traditionnellement dans une boucle assez facile à représenter : il observe le projet, modifie quelques fichiers, exécute des tests, examine ses différences, puis crée un commit. Même lorsqu'il collabore avec une équipe, il connaît généralement l'intention de son propre dossier de travail.

Avec les agents, cette boucle change de dimension. Le même développeur peut confier une correction à un premier agent, une migration à un deuxième et une analyse de performances à un troisième. Chacun peut déléguer une partie de son travail à des sous-agents. En quelques minutes, plusieurs producteurs écrivent du code, ajoutent des tests, déplacent des fichiers ou proposent des commits.

Le développeur n'est plus seulement l'auteur de changements. Il devient responsable d'un petit système de production.

IMAGE MENTALE

De l'établi à l'atelier

Avec un seul développeur, Git ressemble à un établi : les outils et les pièces restent sous les yeux de la personne qui travaille. Avec plusieurs agents, Git ressemble à un atelier comportant plusieurs postes. Chaque poste peut produire une pièce correcte, mais quelqu'un doit distribuer les plans, éviter que deux personnes utilisent la même machine, contrôler les pièces et décider de l'ordre d'assemblage.

Git reste indispensable dans cet atelier. Il conserve les versions, compare les états et relie les commits entre eux. Mais Git ne distribue pas les missions et ne connaît pas l'intention de l'équipe.

Ce qui change réellement

Le changement le plus visible est le volume. Un agent peut produire rapidement un diff important. Le changement le plus dangereux est pourtant moins spectaculaire : plusieurs résultats peuvent être corrects séparément et incompatibles ensemble.

Imaginons quatre missions lancées depuis le même commit :

- l'agent A renomme un service et adapte ses tests ;
- l'agent B ajoute un appel à l'ancien nom dans un nouveau job ;
- l'agent C modifie la structure d'une table utilisée par les deux services ;
- l'agent D met à jour la documentation et un fichier de configuration.

Git détectera peut-être un conflit si deux missions changent les mêmes lignes. Il ne signalera pas forcément que le job de l'agent B appelle désormais une interface supprimée par l'agent A. Il ne saura pas non plus si la migration de l'agent C doit être déployée avant ou après le code.

Le problème n'est donc pas seulement : « plusieurs agents modifient-ils le même fichier ? » La question devient : « plusieurs missions modifient-elles le même système, la même règle ou le même ordre d'exécution ? »

Les trois capacités à ne pas confondre

Pour diriger plusieurs producteurs de code, l'équipe doit séparer trois capacités.

Produire signifie écrire du code, des tests, des migrations ou de la documentation.

Intégrer signifie construire un état combiné, examiner ce résultat et décider qu'il peut rejoindre la branche partagée.

Valider signifie apporter les preuves que ce résultat combiné respecte le besoin, les contraintes et l'état réel de l'application.

Un agent peut être très efficace pour produire. Cette efficacité ne lui donne pas automatiquement l'autorité d'intégrer ni la capacité de valider seul l'ensemble du système.

> **À RETENIR**
>
> **Le débit de code n'est pas le débit de livraison**
>
> Une équipe peut terminer vingt missions et n'intégrer correctement que trois changements. Le stock de branches, de diffs et de décisions en attente est alors plus important que le nombre de tâches annoncées comme terminées.

Le nouveau rôle du développeur responsable

Le développeur responsable d'agents accomplit au moins sept fonctions :

1. il définit l'état de départ de chaque mission ;
2. il attribue un périmètre physique et fonctionnel ;
3. il limite la délégation aux sous-agents ;
4. il conserve la provenance des résultats ;
5. il examine les différences et les preuves ;
6. il organise l'ordre d'intégration ;
7. il décide d'accepter, de reprendre ou d'abandonner un résultat.

Cette responsabilité ne signifie pas qu'il doit relire manuellement chaque caractère. Elle signifie qu'il doit concevoir un processus dans lequel une différence importante, une extension de périmètre ou une preuve absente devient visible avant l'intégration.

Un scénario Tansa, sans avoir besoin de connaître Tansa

Tansa est organisée en modules successifs. Layer1 acquiert des faits Bitcoin. Cluster construit ensuite des regroupements prudents. D'autres modules produisent des profils, des comportements et des catégories. Chaque module dépend d'un état certifié du module précédent.

Supposons qu'un agent optimise Layer1 et qu'un autre modifie en parallèle Cluster. Ils peuvent travailler dans des fichiers entièrement différents. Git peut fusionner leurs branches sans conflit textuel. Pourtant, le second changement peut reposer sur une structure ou une garantie que le premier vient de modifier.

L'enseignement ne concerne pas spécifiquement Bitcoin :

> Deux missions peuvent être en conflit lorsqu'elles touchent la même responsabilité ou le même contrat, même si elles ne partagent aucune ligne de code.

Dans une boutique en ligne, le même problème peut apparaître entre le calcul du prix et la facturation. Dans une application médicale, il peut apparaître entre l'import de données et un moteur de règles. Dans une plateforme logistique, il peut relier le stock et la préparation des commandes.

La première méthode : une fiche par mission

Avant de lancer un agent, le développeur crée une fiche de mission. Il ne s'agit pas d'une longue spécification. C'est une frontière minimale qui permet d'attribuer le travail et de comprendre plus tard ce qui a été produit.

```
mission_id: TANSA-L1-042
owner: victor
agent: agent-layer1
base_commit: 4a1b2c3d4e5f...
branch: agent/layer1-output-audit
worktree: ../worktrees/layer1-output-audit
objective: "Ajouter un contrôle du nombre d'outputs traités"
allowed_paths:
  - app/services/layer1/
  - test/services/layer1/
forbidden_paths:
  - db/migrate/
  - app/services/cluster/
impact_surfaces:
  - layer1_certification_contract
subagents:
  allowed: true
  maximum: 1
tests_required:
  - test/services/layer1/output_audit_test.rb
stop_if:
  - "une migration devient nécessaire"
  - "le contrat lu par Cluster doit changer"
delivery:
  commit_required: false
  report_required: true
```

Les valeurs sont des exemples. Le point important est la relation entre les champs.

Le `base_commit` identifie l'état à partir duquel l'agent raisonne. Les chemins autorisés limitent son espace d'écriture. Les surfaces d'impact décrivent les responsabilités qui peuvent être affectées au-delà des fichiers. Les conditions d'arrêt empêchent une extension silencieuse de la mission.

Avant la première mission : observer sans modifier

Le développeur doit d'abord savoir dans quel dossier et dans quel état il se trouve. Les commandes suivantes sont toutes présentées séparément parce qu'elles répondent à des questions différentes.

Commande 1 — Dans quel dossier suis-je ?

Pourquoi l'utiliser : un terminal peut rester ouvert dans un ancien clone ou dans le worktree d'une autre mission. Cette commande affiche le dossier courant.

Effet exact : `pwd` est une commande du système, pas une commande Git. Elle lit le chemin courant et ne modifie rien.

Sortie attendue : un chemin, par exemple `/projets/tansa`.

Limite : ce chemin ne prouve pas encore qu'il s'agit de la racine du bon dépôt.

Risque : R0 — observation.

```
pwd
```

Exemple :

```
/home/victor/worktrees/layer1-output-audit
```

Cette sortie indique le dossier actuel. Si l'on attendait `/home/victor/tansa`, il faut s'arrêter et comprendre pourquoi le terminal se trouve dans un autre worktree.

Commande 2 — Quel dépôt contient ce dossier ?

Pourquoi l'utiliser : un chemin ressemblant au projet peut être une copie indépendante ou un sous-dossier. Cette commande demande à Git la racine du dossier de travail auquel le terminal appartient.

Effet exact : `git rev-parse --show-toplevel` interroge la structure du dépôt et affiche le chemin de sa racine. Elle ne change ni les fichiers ni les références.

Option expliquée : `--show-toplevel` demande le dossier supérieur du worktree courant.

Sortie attendue : un chemin absolu. Une erreur comme `not a git repository` signifie que Git ne reconnaît aucun dépôt depuis cet emplacement.

Risque : R0 — observation.

```
git rev-parse --show-toplevel
```

Exemple :

```
/home/victor/worktrees/layer1-output-audit
```

Dans un worktree lié, cette racine est le dossier du worktree, pas nécessairement le dossier dans lequel le dépôt a été cloné à l'origine.

Commande 3 — Quelle branche ce dossier affiche-t-il ?

Pourquoi l'utiliser : le nom de branche fournit un repère humain utile pour attribuer le chantier.

Effet exact : `git branch --show-current` lit la branche associée à `HEAD`. Elle ne crée, ne déplace et ne supprime aucune branche.

Option expliquée : `--show-current` demande uniquement le nom court de la branche courante.

Sortie attendue : un nom comme `agent/layer1-output-audit`. Une sortie vide peut indiquer que `HEAD` est détaché : le dossier affiche alors directement un commit sans être positionné sur une branche locale.

Limite : le nom d'une branche est mobile. Il ne remplace pas le hash du commit.

Risque : R0 — observation.

```
git branch --show-current
```

Commande 4 — Quel commit exact est affiché ?

Pourquoi l'utiliser : deux personnes peuvent parler de la même branche à des instants différents. Le hash identifie l'objet commit réellement observé.

Effet exact : `git rev-parse --verify HEAD` résout `HEAD` et vérifie que cette référence existe. Elle ne déplace pas `HEAD`.

Options expliquées : `--verify` exige une référence valide ; `HEAD` désigne le commit actuellement affiché dans ce worktree.

Sortie attendue : un hash complet. Une erreur signifie que la référence n'est pas résolue, par exemple dans un dépôt qui ne possède encore aucun commit.

Risque : R0 — observation.

```
git rev-parse --verify HEAD
```

Exemple :

```
4a1b2c3d4e5f678901234567890abcdef1234567
```

Ce hash peut être copié dans la fiche de mission. Il ne change pas lorsque la branche avance ; il continue de désigner le même commit tant que l'objet reste disponible.

Commande 5 — Quels changements locaux sont déjà présents ?

Pourquoi l'utiliser : lancer un agent dans un dossier déjà modifié mélange les responsabilités. Il faut identifier les différences présentes avant la mission.

Effet exact : `git status --short --branch` compare le commit `HEAD`, l'index et le dossier de travail. Selon la documentation officielle, `git status` montre aussi les fichiers non suivis qui ne sont pas ignorés.

Options expliquées : `--short` produit une sortie compacte ; `--branch` ajoute les informations sur la branche.

Sortie attendue : une première ligne de branche, puis éventuellement des lignes de fichiers. Les deux colonnes de statut représentent respectivement l'index et le dossier de travail.

Limite importante : un état sans fichier modifié signifie seulement qu'aucune différence suivie ou non suivie visible n'est signalée. Cela ne prouve pas que le bon commit, la bonne base de données ou la bonne application est utilisé. Par ailleurs, Git peut actualiser certaines informations mises en cache de l'index pendant `status`; dans des automatisations très concurrentes, la documentation recommande d'examiner l'usage de `--no-optional-locks`.

Risque : R0 — observation fonctionnelle.

```
git status --short --branch
```

Exemple :

```
## agent/layer1-output-audit
 M app/services/layer1/output_audit.rb
?? notes/measurement.txt
```

`M` indique ici une modification du dossier non préparée dans l'index. `??` désigne un fichier non suivi. Avant d'attribuer le dossier à un agent, l'équipe doit identifier le propriétaire de ces deux éléments.

Commande 6 — Quels autres worktrees utilisent ce dépôt ?

Pourquoi l'utiliser : plusieurs agents peuvent déjà travailler dans des dossiers liés au même dépôt. Cette commande construit l'inventaire.

Effet exact : `git worktree list --porcelain` lit les métadonnées des worktrees. Elle ne crée et ne retire aucun dossier.

Option expliquée : `--porcelain` demande un format stable, prévu pour être analysé par un outil. Chaque enregistrement contient notamment le chemin, le hash de `HEAD` et la branche éventuelle.

Sortie attendue : plusieurs blocs séparés par une ligne vide. `detached` indique un `HEAD` détaché ; `locked` un worktree verrouillé ; `prunable` une entrée dont le chemin associé semble ne plus exister.

Limite : la liste dit où se trouvent les worktrees connus. Elle n'attribue pas leurs contenus à un humain ou à un agent et n'explique pas leur mission.

Risque : R0 — observation.

```
git worktree list --porcelain
```

Exemple simplifié :

```
worktree /home/victor/tansa
HEAD 91aa22bb33cc44dd55ee66ff7788990011223344
branch refs/heads/main

worktree /home/victor/worktrees/layer1-output-audit
HEAD 4a1b2c3d4e5f678901234567890abcdef1234567
branch refs/heads/agent/layer1-output-audit
```

Cette photographie établit deux états distincts. Elle ne dit pas encore si la branche de mission part bien du commit attendu ; cette comparaison sera traitée dans un chapitre ultérieur.

Le prompt d'observation initiale

Le prompt suivant autorise uniquement les observations expliquées ci-dessus. Il ne demande pas à l'agent de « préparer » ou de « nettoyer » le dépôt.

```
Tu dois établir l'état de départ de cette mission sans modifier le
dépôt.

Commandes autorisées, et seulement celles-ci :
- pwd : afficher le dossier courant ;
- git rev-parse --show-toplevel : identifier la racine du worktree ;
- git branch --show-current : afficher la branche courante ;
- git rev-parse --verify HEAD : obtenir le hash complet du commit
  affiché ;
- git status --short --branch : inventorier les différences locales
  visibles ;
- git worktree list --porcelain : inventorier les worktrees liés.

N'exécute aucune autre commande. Ne modifie aucun fichier, l'index,
aucune
branche ni référence. Ne crée ni commit ni stash. Ne change pas de
branche.

Présente séparément :
1. les faits observés ;
2. les anomalies par rapport à la mission ;
3. les informations manquantes ;
4. les raisons qui imposent de ne pas commencer.
```

Le prompt ne rend pas les commandes sûres par magie. Il rend l'autorisation compréhensible et vérifiable. L'environnement d'exécution et les permissions doivent, lorsque cela est possible, appliquer les mêmes limites.

Les erreurs fréquentes

Lancer tous les agents depuis `main`. Le nom `main` ne garantit pas que tous démarrent au même hash. La branche peut avancer entre deux lancements.

Utiliser le même dossier pour plusieurs agents. Les agents partagent alors le même index et les mêmes fichiers. Il devient difficile d'attribuer une différence et une commande de changement de branche peut perturber tout le monde.

Confondre mission terminée et résultat intégrable. Un agent peut avoir satisfait son objectif local tout en reposant sur une base ancienne ou en modifiant une responsabilité réservée.

Autoriser les sous-agents sans limite. La délégation peut diluer le périmètre et rendre la provenance difficile à reconstruire.

Demander un commit de sauvegarde. Un commit attribue et enregistre un état. Il ne doit pas servir de réaction automatique à une situation que personne ne comprend encore.

Checklist avant de lancer plusieurs agents

- [ ] Chaque mission possède un identifiant et un responsable humain.
- [ ] Le commit de départ est enregistré par son hash complet.
- [ ] Le dossier de travail est identifié.
- [ ] Les différences préexistantes ont un propriétaire.
- [ ] Les chemins autorisés et interdits sont écrits.
- [ ] Les surfaces fonctionnelles touchées sont déclarées.
- [ ] Le nombre de sous-agents autorisés est défini.
- [ ] Les tests et le rapport attendus sont précisés.
- [ ] Les conditions d'arrêt sont explicites.
- [ ] La voie d'intégration est connue avant la production du code.

Livrable d'équipe : registre des missions actives

```
repository: application
reference_branch: main
observed_reference_commit: FULL_HASH
missions:
  - id: MISSION-001
    owner: HUMAN_OWNER
    agent: AGENT_ID
    worktree: ABSOLUTE_PATH
    branch: agent/mission-001
    base_commit: FULL_HASH
    allowed_paths: []
    impact_surfaces: []
    subagents:
      maximum: 0
    status: ready
    integration_order: undecided
```

Ce registre ne remplace pas Git. Il apporte à Git les informations organisationnelles que Git ne possède pas : mission, propriétaire, limites et décision attendue.

Critères de réussite

Le système est prêt pour le travail multi-agent lorsque chaque résultat futur pourra être relié à une mission, un responsable, un worktree, une branche et un commit de départ. Si l'équipe découvre un fichier modifié sans propriétaire, un agent sans base précise ou un sous-agent inconnu, elle n'augmente pas le parallélisme : elle restaure d'abord la traçabilité.

CHAPITRE 2 — POURQUOI GIT NE COORDONNE PAS AUTOMATIQUEMENT LES AGENTS

Git enregistre les états ; l'équipe organise le travail

Le malentendu de départ

Git est souvent présenté comme un outil de collaboration. Cette formule est vraie, mais elle peut conduire à une attente excessive. Git permet à plusieurs personnes de conserver, comparer et combiner des versions. Il ne sait pas pourquoi ces personnes travaillent, quelles missions dépendent les unes des autres ni quel résultat doit être déployé en premier.

Remplaçons les personnes par des agents et ajoutons des sous-agents : la limite devient immédiatement visible.

Git peut répondre à la question : « quels fichiers diffèrent entre ces deux commits ? » Il ne peut pas répondre seul à la question : « ces deux changements respectent-ils encore le même contrat métier ? »

> **IMAGE MENTALE**
>
> **Git est le registre, pas le chef de chantier**
>
> Un registre peut indiquer quelles pièces ont été reçues, leur version et leur origine. Il ne décide pas quelles équipes doivent les fabriquer, dans quel ordre les assembler ni si le bâtiment obtenu respecte le plan. Ces décisions appartiennent au système de coordination construit autour du registre.

Ce que Git sait réellement

Git représente principalement des objets et des références.

Un blob conserve le contenu d'un fichier. Une tree décrit un répertoire et relie des noms à des objets. Un commit relie une arborescence à un ou plusieurs parents et ajoute des métadonnées. Une référence, comme une branche, contient un nom qui désigne un objet, généralement un commit.

Dans un worktree, Git met également en relation trois états essentiels :

1. le commit désigné par `HEAD` ;
2. l'index, qui prépare le contenu du prochain commit ;
3. le dossier de travail, que les outils et les agents peuvent modifier.

Grâce à ces structures, Git peut calculer des différences, suivre une histoire et construire des fusions.

Ce que Git ne sait pas

Git ne contient pas naturellement :

- l'objectif produit de la mission ;
- le responsable humain de l'agent ;
- le nombre de sous-agents autorisés ;
- les composants réservés par une autre mission ;
- l'ordre requis entre une migration et un déploiement ;
- la signification d'un test métier ;
- l'état de la base de données ;
- la configuration réellement chargée en production ;
- le commit réellement déployé, sauf si l'organisation l'enregistre ;
- la décision d'accepter ou d'abandonner un résultat.

Ce manque n'est pas un défaut de Git. Il définit la frontière de l'outil.

Quatre conflits différents

Lorsque deux agents travaillent en parallèle, le mot « conflit » peut désigner au moins quatre réalités.

1. Le conflit textuel

Deux changements touchent des lignes que Git ne sait pas combiner automatiquement. C'est le conflit le plus visible. Des marqueurs apparaissent et une décision humaine ou assistée est nécessaire.

2. Le conflit sémantique

Les fichiers se combinent, mais les intentions sont incompatibles. Un agent renomme une méthode tandis qu'un autre ajoute un nouvel appel à l'ancien contrat dans un autre fichier. Git peut fusionner les textes et produire un logiciel incorrect.

3. Le conflit temporel

La mission était correcte lorsqu'elle a commencé, mais sa base est devenue trop ancienne. Une autre évolution a changé une hypothèse nécessaire. L'absence de conflit textuel ne rend pas automatiquement le résultat actuel.

4. Le conflit opérationnel

Le code et l'historique sont cohérents, mais l'ordre d'exécution réel est mauvais : migration destructive trop tôt, ancienne instance encore active, variable absente ou données non préparées.

> **À RETENIR**
>
> **Une fusion réussie ne valide qu'une opération Git**
>
> Elle ne prouve ni la compatibilité métier, ni la validité des données, ni la réussite d'un déploiement.

Exemple Tansa : des fichiers différents, une même chaîne de confiance

Dans Tansa, Layer1 fournit des faits certifiés à Cluster. Cluster ne doit pas travailler au-delà du checkpoint fiable de Layer1. Supposons deux missions :

- l'agent A modifie la manière dont Layer1 certifie une hauteur ;
- l'agent B optimise la sélection des blocs autorisés dans Cluster.

Les chemins peuvent être distincts. Pourtant, les missions touchent une même surface : le contrat de certification entre les modules.

La solution ne consiste pas à apprendre à Git ce qu'est un checkpoint Bitcoin. Elle consiste à déclarer cette surface dans les contrats de mission :

```
mission_a:
  allowed_paths:
    - app/services/layer1/
  impact_surfaces:
    - certified_checkpoint_contract

mission_b:
  allowed_paths:
    - app/services/cluster/
  impact_surfaces:
    - certified_checkpoint_contract
```

Un outil d'orchestration ou une revue humaine peut alors détecter le chevauchement logique avant la fusion.

Dans une autre application, `certified_checkpoint_contract` pourrait devenir `payment_state_contract`, `patient_record_version` ou `inventory_reservation_rule`. La méthode reste la même.

La méthode des quatre cartes

Pour compléter ce que Git ne sait pas, l'équipe maintient quatre cartes légères.

La carte des missions relie chaque agent à son objectif, sa base et son responsable.

La carte des espaces relie chaque mission à un worktree, une branche et des chemins autorisés.

La carte des dépendances indique les surfaces partagées et l'ordre entre les missions.

La carte d'intégration indique quel résultat doit être combiné, testé et approuvé.

Ces cartes peuvent vivre dans un fichier YAML, une base interne ou un outil de suivi. L'essentiel est qu'elles soient observables et reliées aux hashes Git.

Observer les relations Git sans les modifier

Les commandes de ce chapitre servent uniquement à comprendre des états existants. Elles ne créent pas de branche et ne fusionnent rien.

Commande 1 — Voir l'histoire récente et les branches visibles

Pourquoi l'utiliser : avant de coordonner des missions, il faut voir comment les commits récents se rattachent les uns aux autres.

Effet exact : `git log --oneline --decorate --graph --all -20` parcourt les commits accessibles par les références locales et les affiche. Elle ne déplace aucune référence.

Options expliquées : `--oneline` condense chaque commit ; `--decorate` montre les noms de références ; `--graph` dessine les relations ; `--all` inclut toutes les références locales connues ; `-20` limite l'affichage à vingt commits.

Sortie attendue : des lignes contenant un symbole de graphe, un hash abrégé, des noms éventuels et un message.

Limites : `--all` montre les références connues dans le dépôt local. Il ne contacte pas le serveur. Une référence `origin/main` peut être ancienne si aucun `fetch` récent n'a été effectué. Le graphe ne montre pas les missions ou dépendances métier.

Risque : R0 — observation.

```
git log --oneline --decorate --graph --all -20
```

Exemple simplifié :

```
* a812fd3 (agent/cluster-window) Optimize cluster window
| * 92ce441 (agent/layer1-audit) Add Layer1 output audit
|/
* 4a1b2c3 (main) Certify pipeline checkpoint
```

Les deux branches visibles partagent ici le même parent `4a1b2c3`. Cette relation ne dit pas si leurs modifications sont compatibles.

Commande 2 — Trouver l'ancêtre commun de deux résultats

Pourquoi l'utiliser : deux branches peuvent avoir des noms clairs mais être parties de bases différentes. L'ancêtre commun permet de situer leur séparation.

Effet exact : `git merge-base A B` calcule un meilleur ancêtre commun entre les deux révisions. Elle ne fusionne pas les branches.

Paramètres expliqués : `A` et `B` doivent être remplacés par deux références ou, de préférence dans un rapport, par deux hashes complets confirmés.

Sortie attendue : un hash de commit. Une erreur ou une sortie absente doit être examinée ; les histoires peuvent ne pas partager d'ancêtre exploitable.

Limite : un ancêtre commun décrit la topologie Git, pas la base déclarée dans le contrat de mission. Une mission peut avoir été recréée ou avoir incorporé d'autres changements.

Risque : R0 — observation.

```
git merge-base A B
```

Exemple :

```
4a1b2c3d4e5f678901234567890abcdef1234567
```

Le rapport peut comparer ce résultat aux `base_commit` déclarés. Une différence n'est pas automatiquement une faute ; elle demande une explication.

Commande 3 — Compter les commits propres à chaque côté

Pourquoi l'utiliser : l'équipe veut une première mesure de l'écart entre deux états.

Effet exact : `git rev-list --left-right --count A...B` parcourt l'histoire et compte les commits accessibles uniquement depuis chaque côté.

Syntaxe expliquée : `A...B` avec trois points demande la différence symétrique ; `--left-right` conserve le côté d'appartenance ; `--count` remplace la liste par deux nombres.

Sortie attendue : deux nombres. Le premier correspond au côté gauche `A`, le second au côté droit `B`.

Limite : le nombre de commits ne mesure ni le nombre de lignes ni la gravité des changements. Un seul commit peut modifier un contrat critique.

Risque : R0 — observation.

```
git rev-list --left-right --count A...B
```

Exemple :

```
2	5
```

`A` possède ici deux commits qui ne sont pas dans `B`, et `B` en possède cinq qui ne sont pas dans `A`. Cette sortie justifie une analyse ; elle ne décide pas quelle branche est correcte.

Commande 4 — Lister les fichiers différents entre une base et un résultat

Pourquoi l'utiliser : la carte des missions déclare des chemins autorisés. Il faut la comparer aux chemins réellement modifiés.

Effet exact : `git diff --name-status BASE...RESULT` liste les chemins et leur type de changement entre le résultat et leur ancêtre commun avec la base.

Options et paramètres : `--name-status` affiche le statut et le chemin sans montrer le contenu ; `BASE` et `RESULT` doivent être remplacés par des hashes ou références vérifiés ; les trois points demandent une comparaison depuis l'ancêtre commun.

Sortie attendue : une lettre puis un chemin. `A` signifie ajouté, `M` modifié, `D` supprimé et `R` renommé avec un score éventuel.

Limites : la commande ne montre pas les fichiers ignorés ou non suivis d'un autre worktree. Elle ne révèle pas non plus les responsabilités logiques touchées par ces fichiers.

Risque : R0 — observation.

```
git diff --name-status BASE...RESULT
```

Exemple :

```
M       app/services/cluster/window_selector.rb
A       test/services/cluster/window_selector_test.rb
```

Si le contrat autorisait seulement `app/services/cluster/` et `test/services/cluster/`, les chemins respectent la frontière physique. La surface fonctionnelle doit encore être examinée.

Commande 5 — Prévisualiser une fusion sans toucher au worktree

Pourquoi l'utiliser : avant d'essayer une fusion dans un dossier, l'équipe peut demander à Git de calculer un résultat potentiel et de signaler des conflits textuels.

Effet exact : `git merge-tree --write-tree A B` calcule une fusion et crée un objet tree représentant le résultat. Elle ne crée aucun commit, ne déplace aucune branche et ne lit ni n'écrit l'index ou le dossier de travail. Elle ajoute néanmoins un objet dans la base d'objets locale : ce n'est donc pas une observation strictement sans écriture interne.

Paramètres : `A` et `B` doivent être remplacés par les deux commits ou branches à comparer. L'équipe doit vérifier la version de Git utilisée, car les formes et options disponibles de `merge-tree` ont évolué.

Sortie attendue : le hash de l'arbre calculé, suivi d'informations de conflit lorsqu'il y en a. La procédure de l'équipe doit documenter la version de Git et le format qu'elle analyse.

Limite majeure : l'absence de conflit textuel ne prouve aucune compatibilité sémantique ou opérationnelle. La commande ne remplace pas la construction et les tests du candidat réel.

Risque : R1 — création d'un objet local sans déplacement de référence.

```
git merge-tree --write-tree A B
```

Si l'équipe ne maîtrise pas le format de sortie de sa version de Git, elle ne doit pas automatiser une décision à partir de ce résultat. Elle peut créer plus tard un worktree d'intégration dédié et y construire le candidat de manière contrôlée.

La marche à suivre avant de paralléliser

1. Enregistrer le hash de la branche de référence.
2. Créer une mission distincte pour chaque résultat attendu.
3. Déclarer les chemins et les surfaces fonctionnelles.
4. Identifier les dépendances et l'ordre éventuel.
5. Attribuer un worktree et une branche par chantier.
6. Définir qui peut déléguer et à combien de sous-agents.
7. Définir la voie d'intégration avant de lancer la production.
8. Observer régulièrement l'écart entre bases et cible.
9. Suspendre les nouvelles missions lorsque la revue ou l'intégration sature.

Prompt de comparaison de deux missions

```
Compare les missions MISSION_A et MISSION_B en lecture seule.

Entrées approuvées :
- commit de MISSION_A : HASH_A ;
- commit de MISSION_B : HASH_B ;
- contrats de mission : PATHS_TO_CONTRACTS.

Tu peux uniquement utiliser les commandes de lecture expliquées dans ce
chapitre : git log, git merge-base, git rev-list, git diff --name-status
et
git merge-tree avec les paramètres fournis.

Ne lance ni merge, rebase, checkout, switch, reset, restore, clean,
stash,
commit, fetch, pull ou push. Ne modifie aucun fichier.

Rapporte séparément :
1. relation d'historique ;
2. chemins modifiés ;
3. surfaces d'impact communes déclarées ;
4. conflits textuels potentiels ;
5. risques sémantiques ou opérationnels à vérifier ;
6. informations manquantes.

Ne décide pas d'intégrer et ne résous aucun conflit.
```

Les erreurs fréquentes

Croire que des branches différentes assurent l'isolation. Si les agents utilisent le même worktree, ils partagent le dossier et l'index. Si leurs branches touchent le même contrat, l'isolation physique ne résout pas la dépendance logique.

Utiliser uniquement une liste de fichiers. Les chemins sont une première barrière, pas une représentation complète de l'architecture.

Laisser l'agent producteur s'intégrer lui-même. Il devient juge de son périmètre, de ses preuves et de ses conflits avec les autres missions.

Actualiser automatiquement toutes les branches. Une base plus récente peut invalider le raisonnement initial. L'actualisation est une nouvelle décision, suivie de nouveaux tests.

Mesurer la réussite au nombre de tâches terminées. La valeur apparaît lorsque le résultat combiné est compris, validé et livré.

Checklist de coordination

- [ ] Les missions possèdent des hashes de départ explicites.
- [ ] Chaque agent et sous-agent est rattaché à un responsable.
- [ ] Les worktrees ne sont pas partagés entre producteurs actifs.
- [ ] Les chemins autorisés sont déclarés.
- [ ] Les surfaces fonctionnelles communes sont déclarées.
- [ ] Les dépendances d'ordre sont visibles.
- [ ] La branche principale n'est pas un espace d'expérimentation.
- [ ] L'intégrateur connaît le candidat exact à construire.
- [ ] Les tests porteront sur le hash combiné.
- [ ] Une mission peut être arrêtée sans être fusionnée.

Livrable d'équipe : carte de coordination

```
reference:
  branch: main
  commit: FULL_HASH

missions:
  MISSION_A:
    owner: HUMAN_A
    result_commit: HASH_A
    paths: []
    impact_surfaces: []
  MISSION_B:
    owner: HUMAN_B
    result_commit: HASH_B
    paths: []
    impact_surfaces: []

dependencies:
  - before: MISSION_A
    after: MISSION_B
    reason: CONTRACT_OR_MIGRATION_REASON

integration:
  owner: HUMAN_INTEGRATOR
  candidate_commit: pending
  isolated_tests: []
  combined_tests: []
  decision: pending
```

Critères de réussite

L'équipe a compris la frontière de Git lorsque personne n'attend d'une fusion automatique qu'elle décide de la validité du produit. Git fournit les objets, les relations et les différences. La coordination ajoute missions, responsabilités, dépendances et critères de validation.

Le chapitre est appliqué avec succès lorsque deux résultats peuvent être comparés sans les modifier, que leurs dépendances non textuelles sont visibles et qu'un responsable humain sait quelle preuve manque avant toute intégration.

POUR ALLER PLUS LOIN

Cet extrait gratuit contient l'avant-propos et les deux premiers chapitres de *Git à l'ère des agents IA*.

L'édition complète développe 65 situations opérationnelles : contrats de mission, worktrees, périmètres, intégration, commandes dangereuses, incidents, récupération et construction du playbook de l'organisation.