CONSTRUIRE TANSA AVEC L'IA

Apprendre l'architecture logicielle en construisant
une application Bitcoin réelle avec l'intelligence artificielle

TOME 1 — CONCEPTION ET DÉVELOPPEMENT

VICTOR PEREZ

Table des matières

À propos de cet extrait

Cet extrait gratuit contient **trois chapitres complets**, choisis pour représenter chacune des grandes parties de Construire Tansa avec l'IA.

Vous allez suivre un même fil conducteur : partir d'un fait Bitcoin, comprendre comment Tansa le transforme en donnée vérifiable, puis découvrir comment le développeur conserve la responsabilité du système lorsqu'il travaille avec une intelligence artificielle.

Vous pouvez lire les chapitres dans l'ordre ou commencer par celui qui correspond le mieux à votre curiosité :

- **Partie I — Comprendre Bitcoin** : apprendre ce qu'une transaction consomme et ce qu'elle crée réellement ;
- **Partie II — Construire Tansa** : découvrir Layer1, la frontière qui transforme les blocs validés en faits auditables ;
- **Partie III — Programmer et architecturer avec l'IA** : comprendre le nouveau rôle du développeur face à une IA capable de produire du code.

Chaque chapitre conserve les scènes concrètes, les analogies, les schémas, les exemples guidés, les exercices et les corrigés de l'édition complète.

> **Une lecture accessible aux débutants**
>
> Vous n'avez pas besoin de connaître Bitcoin, Ruby on Rails ou PostgreSQL avant de commencer. Les termes techniques sont introduits progressivement, et les détails servent toujours une idée que vous pouvez d'abord comprendre simplement.

Partie I — Comprendre Bitcoin

Chapitre 2 — Comprendre une transaction Bitcoin

Inputs, outputs, frais et monnaie rendue

Avant de commencer — une scène concrète

Alice veut payer Paul 70 000 satoshis. Son portefeuille ne trouve pas une ligne de compte contenant exactement cette somme : il doit choisir d'anciennes sorties, les consommer entièrement, créer le paiement et souvent créer une sortie de monnaie rendue.

Cette scène contient le problème que le chapitre va résoudre. Ne cherchez pas encore à mémoriser les noms de classes, de tables ou de commandes. Commencez par comprendre **la responsabilité** : quelle question faut-il trancher, avec quelles données, et quelle conclusion serait excessive ?

> **L'image mentale du chapitre**
>
> Imaginez des tickets de caisse indivisibles. Pour payer 70 €, Alice peut utiliser un ticket de 100 €, remettre 70 € à Paul et récupérer un nouveau ticket de 29 €, le dernier euro représentant les frais.

La carte du chapitre en une ligne

> **repérer les sorties disponibles → choisir les inputs → créer les outputs → calculer les frais → attendre la confirmation**

Lisez cette chaîne une première fois sans vous arrêter. À la fin du chapitre, vous pourrez revenir ici et expliquer chaque flèche avec vos propres mots.

> **Votre droit de ne pas tout retenir**
>
> Pour une première lecture, retenez le mouvement général et les limites du composant. Les détails d'implémentation servent à montrer que l'architecture est réelle ; vous pourrez les relire lorsque vous travaillerez sur le code.

Dire « Alice envoie 0,40 BTC à Paul » est suffisant pour utiliser un portefeuille. Pour comprendre Bitcoin et construire Tansa, cette phrase ne suffit plus.

Bitcoin ne retire pas 0,40 BTC d'un compte appartenant à Alice pour l'ajouter au compte de Paul. Une transaction référence des sorties antérieures, prouve qu'elle satisfait leurs conditions de dépense et crée de nouvelles sorties.

À retenir

À la fin de ce chapitre, le lecteur doit pouvoir :

- décrire une transaction avec ses inputs et ses outputs ;
- expliquer pourquoi un input ne contient pas simplement un montant débité ;
- identifier un output avec le couple `txid:vout` ;
- calculer les frais et la monnaie rendue ;
- distinguer montant des frais et taux de frais ;
- expliquer le rôle du `scriptPubKey`, du `scriptSig` et du witness sans entrer encore dans le détail des scripts ;
- distinguer une transaction non confirmée d'une transaction confirmée ;
- séparer les faits visibles des interprétations économiques ;
- expliquer ce que Layer1 doit conserver pour Tansa.

---

1. La question de départ

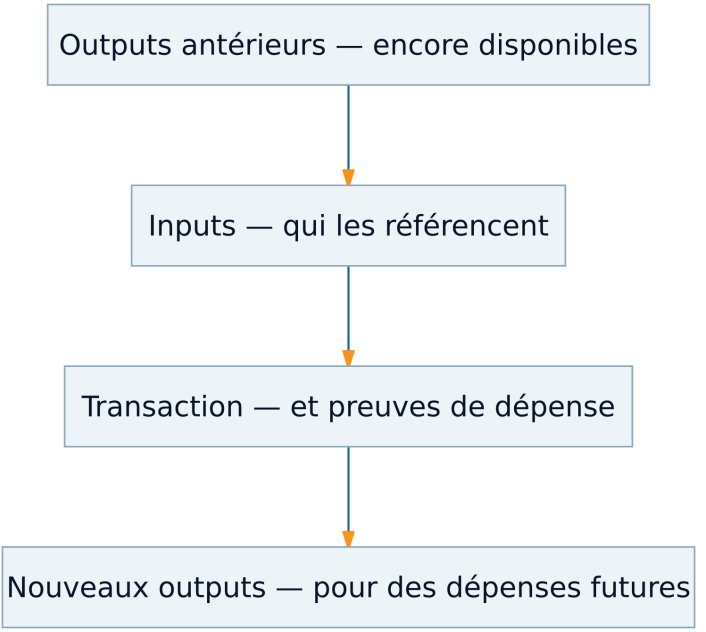
Alice veut payer Paul. Son portefeuille affiche un solde suffisant. Elle saisit un montant, confirme l'opération et voit apparaître un identifiant de transaction.

Que vient-il réellement de se passer ?

La représentation bancaire suggère trois étapes :

1. le compte d'Alice est débité ;
2. le compte de Paul est crédité ;
3. le système central enregistre les nouveaux soldes.

Bitcoin utilise un autre modèle :



À retenir

Une transaction Bitcoin ne déplace pas une ligne de solde. Elle consomme des outputs antérieurs et crée de nouveaux outputs.

2. L'image mentale des coffres

À retenir

Imaginons des coffres transparents posés sur une place publique.

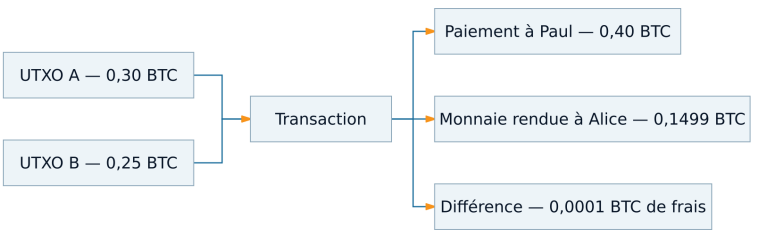
- Chaque coffre contient une valeur précise.
- Chaque coffre possède une serrure définissant les conditions d'ouverture.
- Tout le monde peut voir que le coffre existe et connaître sa valeur.
- Celui qui veut utiliser son contenu doit satisfaire la serrure.
- Un coffre ouvert est consommé entièrement.
- La transaction place ensuite la valeur dans de nouveaux coffres.

Cette analogie aide à comprendre les UTXO. Elle ne doit pas être prise au pied de la lettre : aucun coffre physique n'existe et la condition de dépense est un programme Bitcoin, pas une serrure mécanique.

Alice contrôle deux coffres :

- un output de 0,30 BTC ;
- un output de 0,25 BTC.

Elle souhaite payer Paul 0,40 BTC. Le total disponible dans les deux coffres est de 0,55 BTC. Les deux sont consommés et de nouveaux outputs sont créés.



Les deux anciens outputs totalisent 0,55 BTC. Les nouveaux outputs totalisent 0,5499 BTC. La différence de 0,0001 BTC correspond aux frais.

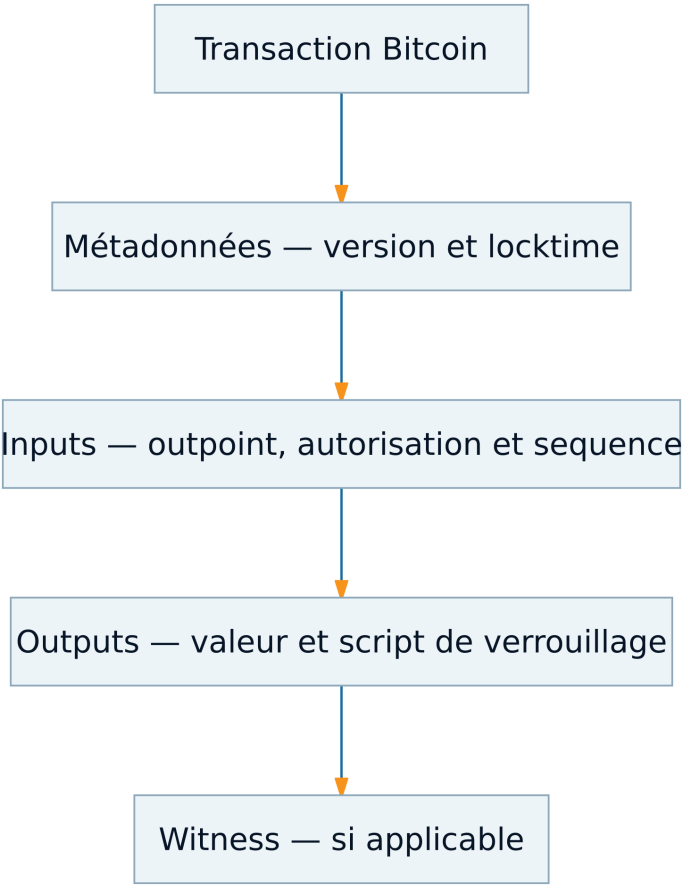
---

3. L'anatomie générale d'une transaction

Une transaction contient principalement :

- une version ;
- une liste d'inputs ;
- une liste d'outputs ;
- un champ `locktime` ;

- et, pour les transactions concernées, des données de witness.



À ce stade, il n'est pas nécessaire de mémoriser la sérialisation binaire. Il faut comprendre les responsabilités.

Input

Un **input** est une entrée de transaction. Dans une transaction ordinaire, il référence un output antérieur et fournit les éléments nécessaires pour en autoriser la dépense.

Output

Un **output** est une sortie créée par une transaction. Il associe une valeur en satoshis à une condition de dépense future.

UTXO

Un **UTXO** est un output qui n'a pas encore été dépensé. L'acronyme signifie Unspent Transaction Output, ou sortie de transaction non dépensée.

---

4. Un output prépare une dépense future

Un output contient deux éléments fondamentaux :

1. une valeur exprimée en satoshis ;
2. un script de verrouillage, généralement appelé `scriptPubKey`.

Le script décrit la condition qui devra être satisfaite pour dépenser cet output plus tard.

À retenir

L'output ressemble à un coffre nouvellement créé. Sa valeur indique ce qu'il contient. Son `scriptPubKey` définit la serrure. La future transaction qui voudra le dépenser devra fournir les éléments permettant de satisfaire cette serrure.

Une adresse est une représentation pratique utilisée par les portefeuilles pour construire certains types de conditions de paiement. Elle n'est pas l'output lui-même.

À retenir

La blockchain contient des transactions, des valeurs et des scripts. Elle ne contient pas naturellement une table « adresse, propriétaire, solde ». Réduire chaque output à « de l'argent appartenant à une personne identifiée » introduirait une information absente du protocole.

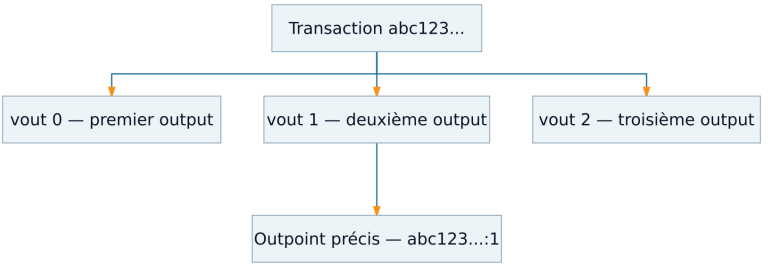
---

5. Identifier précisément un output avec `txid:vout`

Pour dépenser un output, un input doit pouvoir le désigner sans ambiguïté.

Il utilise :

- le `txid` de la transaction qui a créé l'output ;
- le numéro de position de cet output, appelé `vout`.



Le premier output porte l'index `0`, le deuxième l'index `1`, puis la numérotation continue.

> **Outpoint**
>
> Un **outpoint** est la référence précise vers un output antérieur. Il est formé d'un `txid` et d'un index `vout`.
>
> `outpoint = txid:vout`

Dans notre exemple, `abc123...:1` désigne le deuxième output de la transaction `abc123...`.

---

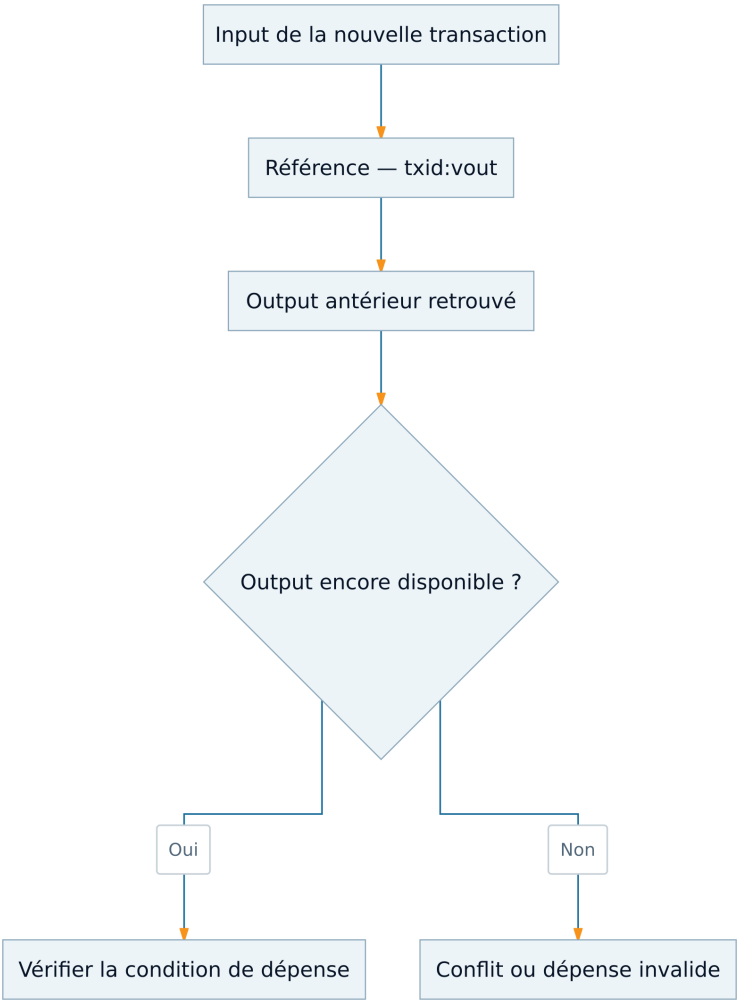
6. Un input ne transporte pas simplement un montant

Un input ne dit pas « débiter 0,30 BTC du compte d'Alice ». Il pointe vers un output existant.

Pour connaître sa valeur, le nœud doit retrouver l'output référencé. Cet output antérieur contient le montant et la condition de dépense.

Le nœud vérifie notamment :

- que l'output référencé existe ;
- qu'il n'a pas déjà été dépensé dans l'état retenu ;
- que la transaction fournit les éléments satisfaisant sa condition ;
- que les autres règles applicables sont respectées.



À retenir

La valeur d'un input vient de l'output qu'il référence. Une application qui ne résout pas correctement cette relation ne peut pas calculer de façon fiable les valeurs dépensées ni les frais.

7. `scriptSig`, witness et autorisation

Pour dépenser un output, la transaction doit fournir les données attendues par sa condition de dépense.

Selon le type de script :

- certaines données se trouvent dans le `scriptSig` de l'input ;
- avec SegWit, les signatures et autres éléments d'autorisation

peuvent se trouver dans le witness ;
- certaines constructions utilisent une combinaison prévue par les règles correspondantes.

Witness

Le **witness**, ou témoin, est une structure introduite par SegWit. Elle contient des données nécessaires à la validation de certaines dépenses, notamment des signatures, tout en les séparant de la sérialisation traditionnelle utilisée pour calculer le `txid`.

À retenir

La clé privée n'est jamais envoyée au réseau. Le portefeuille produit une signature. Les nœuds utilisent les données publiques et les règles du script pour vérifier l'autorisation sans découvrir la clé privée.

Le chapitre consacré aux scripts reprendra cette mécanique avec des exemples P2PKH, SegWit et Taproot. Ici, l'idée centrale est suffisante : un input référence un output et apporte une preuve satisfaisant sa condition.

---

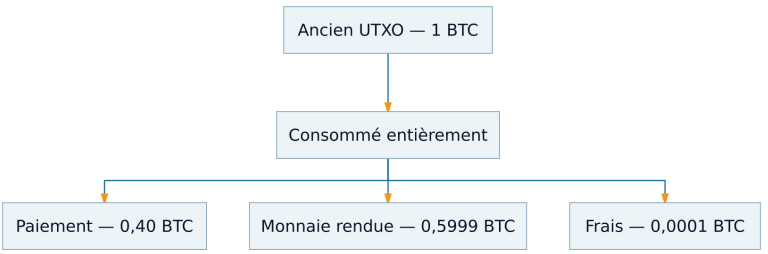
8. Un UTXO est consommé entièrement

Alice contrôle un UTXO de 1 BTC et veut payer 0,40 BTC.

Bitcoin ne retire pas 0,40 BTC de cet UTXO en laissant 0,60 BTC à l'intérieur. L'UTXO de 1 BTC est consommé entièrement.

La transaction crée généralement :

- un output de 0,40 BTC pour le paiement ;
- un output proche de 0,60 BTC pour la monnaie rendue ;
- la différence entre l'input et les outputs devient les frais.



> **À retenir**
> Une transaction ne modifie pas un UTXO existant. Elle le dépense et crée de nouveaux outputs.

---

9. Exemple complet : Alice paie Paul

Alice souhaite payer Paul 0,40 BTC. Son portefeuille sélectionne deux UTXO.

UTXO sélectionné	Valeur
`txid_A:0`	0,30 BTC
`txid_B:1`	0,25 BTC
Total des inputs	**0,55 BTC**

La transaction crée deux outputs.

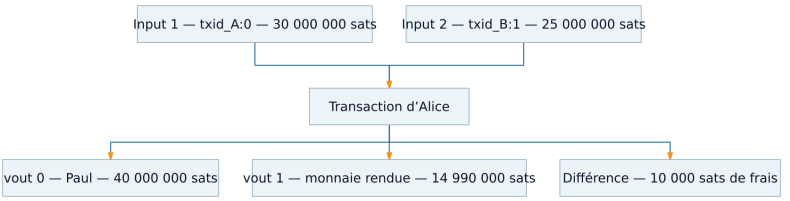
Output créé	Valeur
Paiement destiné à Paul	0,40 BTC
Monnaie rendue au portefeuille d'Alice	0,1499 BTC
Total des outputs	**0,5499 BTC**

Les frais sont calculés par différence :

```
0,55 BTC - 0,5499 BTC = 0,0001 BTC
```

En satoshis :

```
55 000 000 sats - 54 990 000 sats = 10 000 sats
```



La transaction ne contient généralement aucun champ déclarant :

- « Alice est la propriétaire de ces inputs » ;
- « Paul est la personne derrière cet output » ;
- « le premier output est le paiement » ;
- « le second output est la monnaie rendue ».

Ces rôles sont connus du portefeuille ou déduits par un observateur. Ils ne doivent pas être confondus avec les données brutes.

---

10. Les frais ne sont pas un output envoyé au mineur

Dans une transaction ordinaire, les frais ne sont pas représentés par un output explicitement adressé au mineur.

```
frais = somme des valeurs des inputs - somme des valeurs des outputs
```

Le mineur qui inclut la transaction peut réclamer ces frais dans la transaction coinbase du bloc.

> **Montant des frais**
>
> Le **montant des frais** est la différence totale exprimée en satoshis entre les inputs et les outputs.

> **Taux de frais**
>
> Le **taux de frais** relie les frais à la taille virtuelle de la transaction. Il est couramment exprimé en satoshis par octet virtuel, ou `sat/vB`.

Une transaction qui déplace 100 BTC n'est pas nécessairement plus chère qu'une transaction qui déplace 0,01 BTC. Le coût dépend surtout de l'espace occupé et de la demande pour l'espace limité des blocs.

Exemple de taux de frais

Une transaction possède une taille virtuelle de 200 vB et choisit un taux de 15 sat/vB.

```
200 vB x 15 sat/vB = 3 000 satoshis
```

> **À retenir**
>
> Le montant économique transféré et le poids technique de la transaction sont deux grandeurs différentes. Multiplier le montant envoyé ne multiplie pas automatiquement les frais.

---

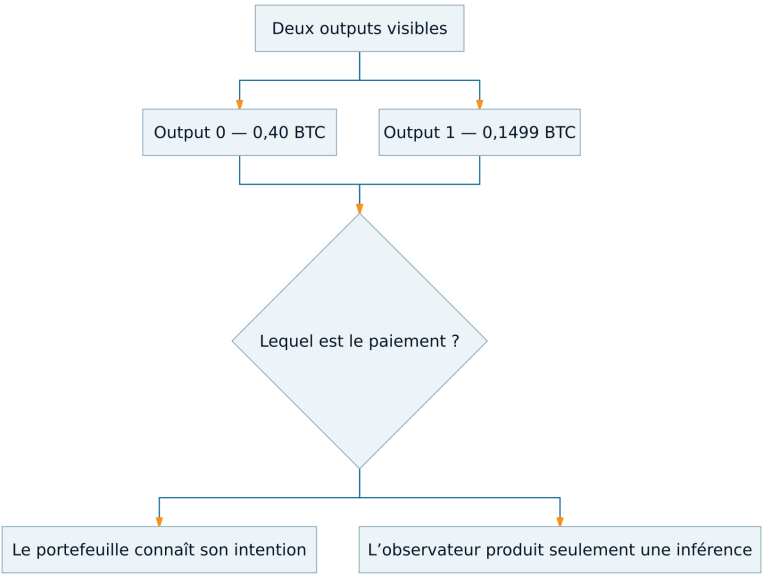
11. La monnaie rendue n'est pas étiquetée

Le portefeuille d'Alice sait quel output il a créé pour récupérer le surplus. Un observateur de la blockchain ne reçoit pas nécessairement cette information.

Il peut chercher des indices :

- type de script ;
- réutilisation ou nouveauté d'une adresse ;
- montant ;
- comportement ultérieur ;
- conventions connues d'un portefeuille.

Mais ces indices peuvent se tromper.



À retenir

La présence et la valeur des outputs sont des faits. Leur rôle économique, paiement ou monnaie rendue, peut être une inférence.

---

12. La sélection des pièces

Le portefeuille doit choisir quels UTXO dépenser. Cette opération est appelée **sélection des pièces**, ou coin selection.

Le choix peut chercher à :

- réduire la taille de la transaction ;
- éviter une monnaie rendue trop faible ;
- préserver la confidentialité ;
- consolider de nombreux petits UTXO ;
- éviter de relier inutilement certaines activités ;
- appliquer la stratégie propre au portefeuille.

À retenir

Si vous devez payer 37 euros avec plusieurs billets et pièces, différentes combinaisons sont possibles. Vous pouvez minimiser le nombre d'éléments utilisés, recevoir de la monnaie ou profiter de l'occasion pour vous débarrasser de petites pièces. Un portefeuille Bitcoin effectue une décision comparable, mais avec des contraintes de frais, de scripts et de confidentialité.

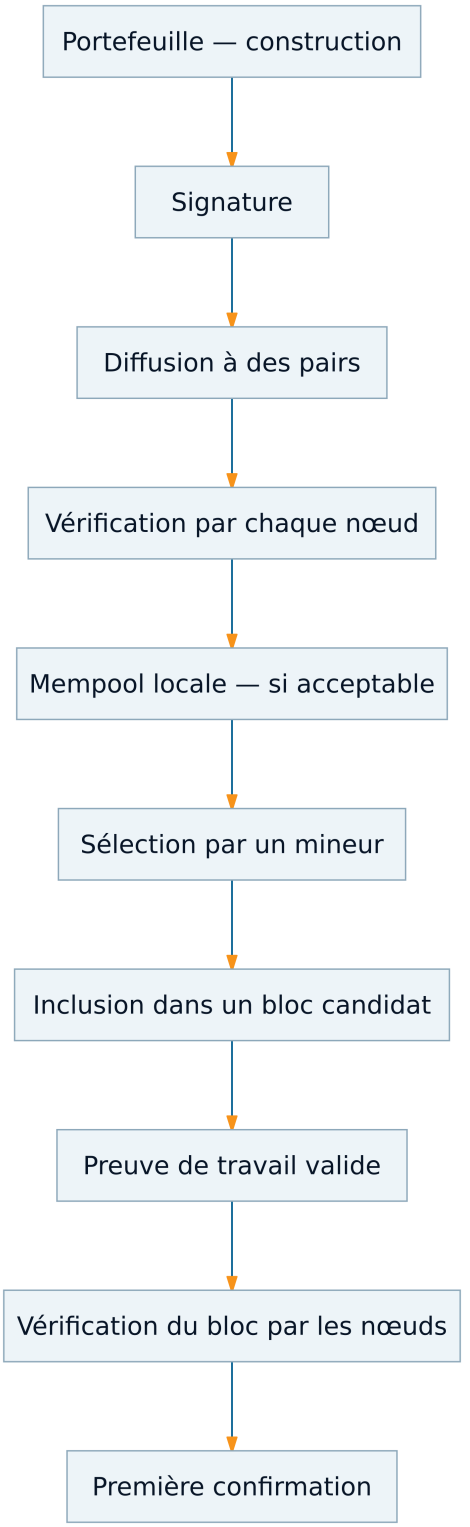
Deux portefeuilles peuvent donc construire des transactions différentes pour réaliser le même paiement.

> **À retenir**
> Une transaction possédant de nombreux inputs ne révèle pas automatiquement plusieurs personnes. Elle peut représenter une consolidation, une stratégie de sélection des pièces, une activité de service ou une construction collaborative.

---

13. De la construction à la confirmation

Une transaction suit plusieurs états avant de devenir un fait confirmé dans un bloc.


Portefeuille — construction
Signature
Diffusion à des pairs
Vérification par chaque nœud
Mempool locale — si acceptable
Sélection par un mineur
Inclusion dans un bloc candidat
Preuve de travail valide
Vérification du bloc par les nœuds
Première confirmation

Construction

Le portefeuille sélectionne des UTXO, crée les outputs, fixe les paramètres et produit les signatures.

Diffusion

La transaction est envoyée à un ou plusieurs pairs, qui peuvent la vérifier puis la relayer.

Mempool

Chaque nœud peut conserver localement la transaction s'il la juge acceptable selon les règles et politiques qu'il applique.

Bloc

Un mineur peut sélectionner la transaction. Après diffusion du bloc, les nœuds vérifient indépendamment toutes les conditions nécessaires.

Confirmation

La transaction obtient sa première confirmation lorsqu'elle est incluse dans la chaîne retenue par le nœud. Sa profondeur augmente lorsque de nouveaux blocs sont ajoutés au-dessus.

---

14. Consensus et politique de mempool

Toutes les règles ne jouent pas le même rôle.

Les **règles de consensus** déterminent si une transaction peut faire partie d'un bloc que le nœud considère valide.

Les **politiques de mempool et de relais** déterminent ce qu'un nœud accepte de conserver et de relayer avant confirmation. Elles peuvent être plus restrictives et varier selon la configuration ou la version du logiciel.

> **À retenir**
>
> Une transaction valide au regard du consensus n'est pas nécessairement acceptée dans la mempool de tous les nœuds. Inversement, la présence dans une mempool ne garantit pas la confirmation future.

Une transaction non confirmée peut :

- rester en attente ;
- être remplacée dans certaines conditions ;
- être supprimée d'une mempool ;
- entrer en conflit avec une autre dépense ;
- être confirmée ultérieurement.

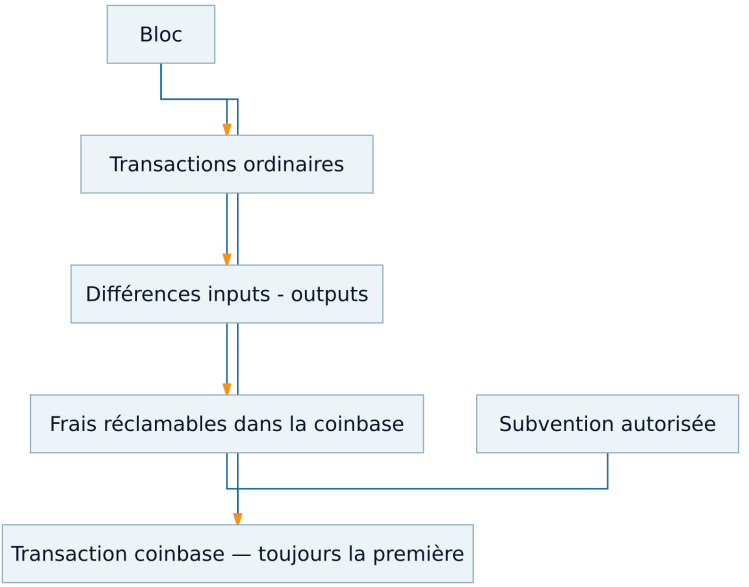
Tansa doit donc distinguer les observations de mempool des faits certifiés à partir des blocs.

15. La transaction coinbase, un cas particulier

La première transaction de chaque bloc est appelée **transaction coinbase**.

Elle ne dépense pas un UTXO antérieur au moyen d'un input ordinaire. Elle permet au mineur de réclamer :

- la subvention de bloc autorisée ;
- les frais des transactions incluses.



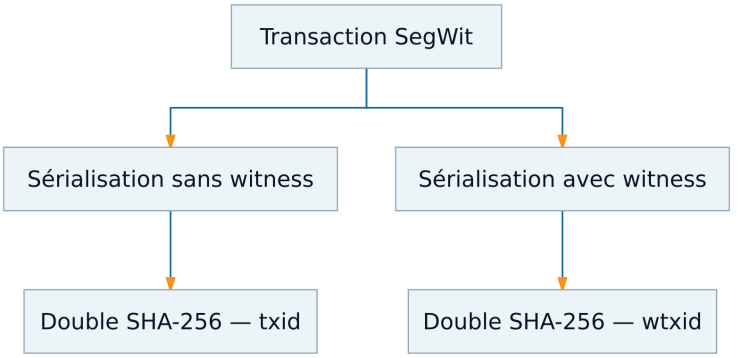
Les outputs de la coinbase possèdent une maturité : ils ne peuvent pas être dépensés avant le nombre de blocs prévu par les règles.

> **À retenir**
> La transaction **coinbase** est un mécanisme du protocole Bitcoin. **Coinbase** est aussi le nom d'une entreprise. Les deux notions ne doivent pas être confondues.

16. `txid`, `wtxid` et SegWit

Le `txid` est l'identifiant traditionnel d'une transaction. Pour une transaction SegWit, les données du witness ne participent pas au calcul du `txid`.

Le `wtxid` identifie la sérialisation qui inclut le witness.



Identifiant calculé à partir de la sérialisation traditionnelle de la transaction. Les références habituelles vers les outputs utilisent `txid:vout`.

Identifiant d'une transaction SegWit calculé à partir de la sérialisation comprenant les données de witness.

À retenir

Pour suivre les dépenses dans Layer1, la relation fondamentale reste : un input désigne un output précis avec son `txid` et son `vout`.

---

17. Ce qu'un observateur sait et ce qu'il suppose

Considérons la transaction d'Alice et Paul.

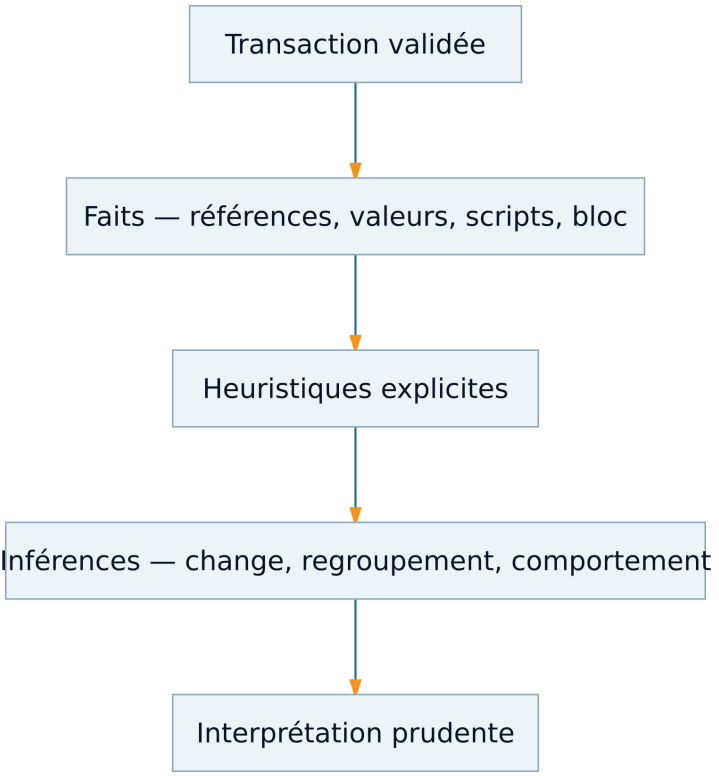
Faits directement observables

- deux inputs référencent deux outputs antérieurs ;
- deux nouveaux outputs sont créés ;
- leurs valeurs sont visibles ;
- la différence correspond aux frais ;
- la transaction est incluse à une hauteur donnée si elle est confirmée.

Informations non inscrites directement

- l'identité civile d'Alice ;
- l'identité civile de Paul ;
- le rôle exact de chaque output ;
- la raison économique du paiement ;

- la propriété commune ou séparée des inputs.



À retenir

La transaction est publique. Son histoire économique complète ne l'est pas nécessairement.

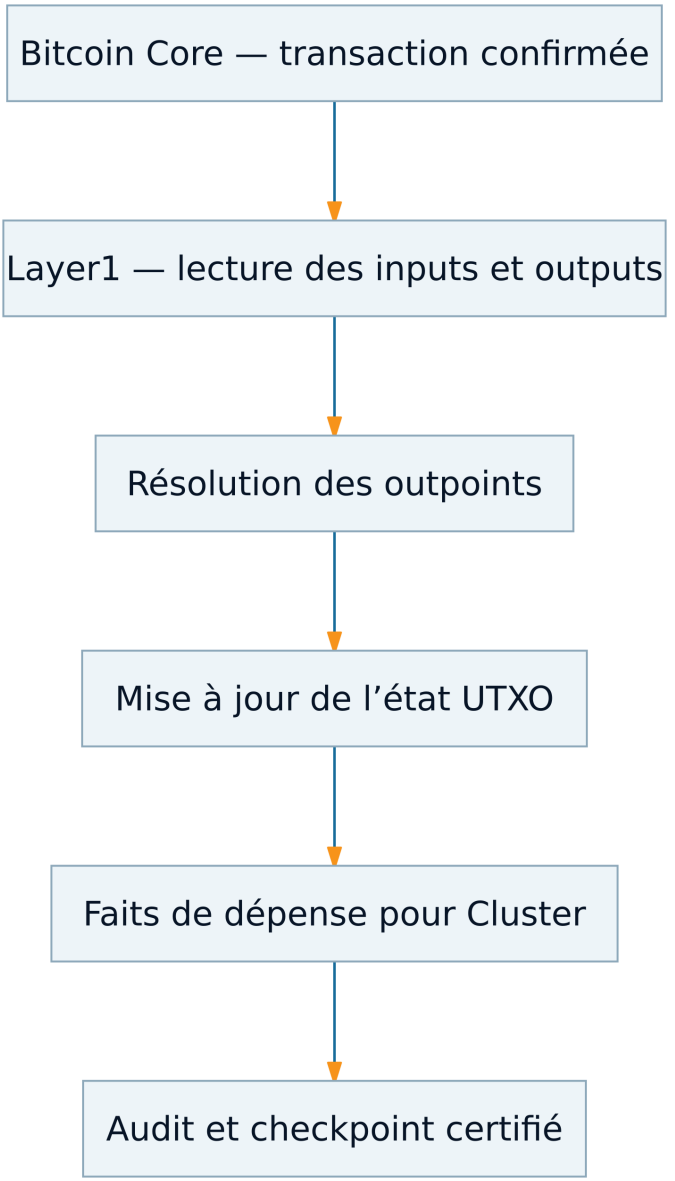
---

18. Ce que Layer1 doit conserver pour Tansa

Lorsqu'une transaction confirmée est lue depuis Bitcoin Core, Layer1 transforme son contenu en faits structurés.

Élément	Rôle dans Tansa
`txid`	Identifier la transaction
hauteur et hash du bloc	Situer la transaction dans une chaîne précise
`vout`	Identifier chaque output
`value_sats`	Conserver la valeur entière en satoshis

Élément	Rôle dans Tansa
script ou représentation dérivée	Préserver la condition de dépense et ce qui peut en être extrait
outpoint de chaque input	Relier la dépense à l’output antérieur
état dépensé	Faire évoluer l’état UTXO
informations d’audit	Prouver que la transformation est complète et cohérente



Layer1 peut certifier les références et les valeurs. Il ne doit pas inventer le rôle économique des outputs.

À retenir

« Output de monnaie rendue » est une interprétation utile lorsqu’elle est connue du portefeuille ou estimée par une règle. « Output numéro 1 de 14 990 000 satoshis » est un fait directement représenté par la transaction.

---

19. Confusions fréquentes

« Un input est un débit »

Un input référence un output antérieur. Sa valeur vient de cet output. La comparaison avec un débit masque la relation entre transactions.

« Un output est forcément reçu par une autre personne »

Un output peut représenter un paiement, de la monnaie rendue, un transfert interne ou une construction technique.

« Les frais sont un troisième output »

Les frais correspondent à la différence entre la somme des inputs et celle des outputs. Le schéma dessine parfois cette différence pour la rendre visible, mais la transaction ordinaire ne crée pas un output de frais destiné au mineur.

« Une transaction non confirmée est définitive »

Sa présence dans une mempool locale ne garantit ni sa présence chez tous les nœuds ni sa confirmation.

« Une adresse est un portefeuille ou une personne »

Un portefeuille peut gérer de nombreuses clés et adresses. Une adresse ne constitue pas une identité civile.

---

20. Ce qu’il faut retenir

À retenir

1. Une transaction ordinaire référence des outputs antérieurs dans ses inputs.

2. Un outpoint est identifié par `txid:vout`.
3. La valeur de l'input vient de l'output référencé.
4. Un output associe une valeur à une condition de dépense future.
5. Un UTXO dépensé est consommé entièrement.
6. La monnaie rendue est un nouvel output.
7. Les frais sont la différence entre la somme des inputs et celle des outputs.
8. Le taux de frais dépend de la taille virtuelle de la transaction, pas directement du montant transféré.
9. Une mempool est locale et une transaction non confirmée reste incertaine.
10. Les nœuds vérifient les transactions et les blocs selon leurs règles.
11. La transaction coinbase est un cas particulier qui réclame la subvention et les frais du bloc.
12. SegWit distingue le `txid` du `wtxid` pour les transactions possédant un witness.
13. La blockchain expose les données techniques, pas automatiquement l'identité, le paiement ou la monnaie rendue.
14. Layer1 doit préserver les faits et laisser les interprétations aux modules appropriés.

---

21. Vérifier sa compréhension

Questions

Atelier guidé — raisonner comme l'architecte de Tansa

Voici une méthode simple pour aborder un exemple de ce chapitre :

1. Lister ce qui est consommé et ce qui est créé.
2. Vérifier que la somme des inputs couvre les outputs.
3. Ne jamais décider trop vite quel output est la monnaie rendue.

Mini-défi

Une transaction consomme 120 000 satoshis et crée deux outputs de 70 000 et 48 000 satoshis. Où sont les frais ?
Prenez quelques secondes pour formuler une réponse. Une phrase précise vaut mieux qu'un long paragraphe rempli de suppositions.

Corrigé raisonné

Ils correspondent à la différence : 120 000 - 70 000 - 48 000 = 2

000 satoshis. Il n'existe pas nécessairement un output nommé « frais ».
Le but n'est pas seulement d'obtenir la bonne réponse. Il est de séparer ce qui est observé, ce qui est calculé et ce qui reste une hypothèse.

Pause — trois idées à garder

- Un input référence une ancienne sortie.
- Un UTXO est consommé entièrement.
- Les frais sont une différence, pas une destination ordinaire.

À retenir

1. Quelle différence existe entre un input et l'output qu'il référence ?
2. Que signifie le couple `txid:vout` ?
3. Pourquoi un UTXO de 1 BTC ne devient-il pas simplement un UTXO de 0,6 BTC après un paiement de 0,4 BTC ?
4. Où la valeur d'un input est-elle trouvée ?
5. Comment calcule-t-on les frais d'une transaction ?
6. Pourquoi le montant transféré ne détermine-t-il pas directement les frais ?
7. La blockchain indique-t-elle explicitement quel output représente la monnaie rendue ?
8. Une transaction présente dans une mempool est-elle confirmée ?
9. Quel est le rôle particulier de la transaction coinbase ?
10. Quelle différence existe entre `txid` et `wtxid` pour une transaction SegWit ?
11. Pourquoi Layer1 doit-il conserver les outpoints ?
12. Peut-on conclure qu'une transaction à trois inputs implique trois personnes ?

Réponses commentées

1. Input et output référencé

L'output antérieur contient la valeur et la condition de dépense. L'input appartient à la nouvelle transaction : il désigne cet output et fournit les données nécessaires à son autorisation.

2. `txid:vout` Le `txid` identifie la transaction créatrice et le `vout` indique la position de l'output dans cette transaction. Ensemble, ils forment un outpoint précis.

3. Consommation entière Bitcoin ne modifie pas la valeur d'un UTXO existant. L'UTXO de 1 BTC est dépensé. La transaction crée un output de paiement, un éventuel output de monnaie rendue et laisse une différence pour les frais.

4. Valeur d'un input Le nœud retrouve l'output référencé par l'outpoint. C'est cet output antérieur qui contient la valeur.

5. Calcul des frais On additionne la valeur de tous les inputs, puis on soustrait la valeur de tous les outputs. Le résultat constitue le montant des frais.

6. Montant transféré et frais Les frais dépendent surtout de la taille virtuelle de la transaction et du taux choisi en sat/vB. Une transaction comportant de nombreux inputs peut être plus volumineuse même si elle transfère peu de valeur.

7. Identification de la monnaie rendue Non. Le portefeuille connaît généralement son propre output de change, mais un observateur doit l'inférer à partir d'indices qui peuvent être trompeurs.

8. Mempool et confirmation Non. La mempool contient localement des transactions non confirmées. La première confirmation arrive lorsque la transaction est incluse dans un bloc de la chaîne retenue par le nœud.

9. Transaction coinbase Elle est la première transaction du bloc et permet au mineur de réclamer la subvention autorisée ainsi que les frais des transactions incluses. Elle ne dépense pas un outpoint ordinaire.

10. `txid` et `wtxid` Pour une transaction SegWit, le `txid` est calculé sans les données de witness, tandis que le `wtxid` couvre la sérialisation qui les inclut.

11. Outpoints dans Layer1 Ils relient chaque dépense à l'output précis qui lui fournit sa valeur et sa condition. Sans cette relation, Tansa ne pourrait pas reconstruire correctement l'état UTXO ni les flux de dépense.

12. Trois inputs, trois personnes ? Non. Les trois inputs peuvent être contrôlés par un seul portefeuille, par plusieurs participants ou provenir d'une construction collaborative. Le nombre d'inputs est un fait ; le nombre de personnes serait une interprétation.

22. Exercices

Exercice 1 - Paiement et monnaie rendue

À retenir

Un portefeuille contrôle un seul UTXO de 0,8 BTC. Il veut payer 0,3 BTC avec 0,0001 BTC de frais.

1. Quelle valeur d'input est consommée ?
2. Quelle valeur reçoit l'output de paiement ?
3. Quelle valeur reçoit l'output de monnaie rendue ?

Correction de l'exercice 1

À retenir

L'unique input référence l'UTXO entier de 0,8 BTC.

```
Input consommé :  0,8000 BTC
Paiement :        0,3000 BTC
Frais :           0,0001 BTC
Monnaie rendue :  0,4999 BTC
```

Le calcul de la monnaie rendue est :

```
0,8000 - 0,3000 - 0,0001 = 0,4999 BTC
```

Exercice 2 - Calculer des frais en satoshis

À retenir

Une transaction dépense deux outputs :

- 120 000 sats ;
- 85 000 sats.

Elle crée deux outputs :

- 150 000 sats ;
- 52 500 sats.

Calculez le montant des frais.

Correction de l'exercice 2

À retenir

Somme des inputs :

```
120 000 + 85 000 = 205 000 sats
```

Somme des outputs :

```
150 000 + 52 500 = 202 500 sats
```

Frais :

```
205 000 - 202 500 = 2 500 sats
```

Exercice 3 - Distinguer fait et interprétation

À retenir

Tansa observe une transaction confirmée possédant trois inputs et deux outputs. L'un des outputs vaut 2 BTC.
Classez les affirmations suivantes : **fait**, **inférence possible** ou **conclusion injustifiée**.

1. La transaction possède trois inputs.
2. Trois personnes ont envoyé des bitcoins.
3. Un output de 2 BTC a été créé.
4. L'output de 2 BTC est une vente.
5. L'un des deux outputs peut être de la monnaie rendue.

Correction de l'exercice 3

À retenir

1. **Fait** : le nombre d'inputs est directement observable.
2. **Conclusion injustifiée** : les inputs ne correspondent pas automatiquement à des personnes distinctes.
3. **Fait** : la valeur de l'output est directement représentée.
4. **Conclusion injustifiée** : une vente ne peut pas être déduite de la valeur seule.
5. **Inférence possible** : un output peut être de la monnaie rendue, mais ce rôle doit être établi par le portefeuille ou estimé avec prudence.

23. Termes introduits dans ce chapitre

À retenir

- **Coin selection** : choix des UTXO utilisés pour construire une transaction.
- **Confirmation** : inclusion d'une transaction dans un bloc de la chaîne retenue.
- **Input** : entrée qui référence généralement un output antérieur et fournit les données d'autorisation.
- **Montant des frais** : différence entre la somme des inputs et celle des outputs.
- **Outpoint** : référence précise `txid:vout` vers un output antérieur.
- **Output** : valeur et condition de dépense créées par une transaction.
- **`scriptPubKey`** : script de verrouillage associé à un output.
- **`scriptSig`** : données placées dans l'input de certaines dépenses pour satisfaire un script.
- **Taux de frais** : frais rapportés à la taille virtuelle, généralement en sat/vB.
- **Transaction coinbase** : première transaction d'un bloc, utilisée pour réclamer subvention et frais.
- **`txid`** : identifiant traditionnel d'une transaction.
- **UTXO** : output non encore dépensé.
- **`vout`** : index d'un output dans sa transaction créatrice.
- **Witness** : structure SegWit contenant des données nécessaires à la validation de certaines dépenses.
- **`wtxid`** : identifiant couvrant la sérialisation comprenant le witness.

---

24. Transition vers le modèle UTXO

Nous savons maintenant suivre une transaction : elle référence des outputs antérieurs, les consomme et crée de nouveaux outputs.

Le chapitre suivant changera d'échelle. Au lieu d'observer une seule transaction, nous examinerons l'ensemble de tous les outputs encore disponibles.

> Comment l'état UTXO évolue-t-il bloc après bloc, et pourquoi un nœud en a-t-il besoin pour valider les nouvelles dépenses ?

---

25. Sources techniques de référence

- Bitcoin Developer Guide - Transactions
- Bitcoin Developer Reference - Transactions
- Bitcoin Developer Guide - Block Chain

- Bitcoin Developer Guide - Mining
- BIP 141 - Segregated Witness
- BIP 125 - Opt-in Full Replace-by-Fee Signaling

Partie II — Construire Tansa

Chapitre 8 — Comprendre Layer1

De Bitcoin Core aux faits certifiés de Tansa

Avant de commencer — une scène concrète

Bitcoin Core vient d'accepter un nouveau bloc. Tansa ne peut pas immédiatement annoncer qu'une baleine accumule : elle doit d'abord transformer ce bloc en faits locaux complets et auditables.

Cette scène contient le problème que le chapitre va résoudre. Ne cherchez pas encore à mémoriser les noms de classes, de tables ou de commandes. Commencez par comprendre **la responsabilité** : quelle question faut-il trancher, avec quelles données, et quelle conclusion serait excessive ?

> **L'image mentale du chapitre**
> Layer1 est la salle des scellés de Tansa. Chaque pièce reçue est enregistrée, numérotée, contrôlée et reliée à son origine avant de quitter la pièce.

La carte du chapitre en une ligne

lire le bloc → construire les faits → résoudre les dépenses → écrire → auditer → certifier le checkpoint

Lisez cette chaîne une première fois sans vous arrêter. À la fin du chapitre, vous pourrez revenir ici et expliquer chaque flèche avec vos propres mots.

> **Votre droit de ne pas tout retenir**
> Pour une première lecture, retenez le mouvement général et les limites du composant. Les détails d'implémentation servent à montrer que l'architecture est réelle ; vous pourrez les relire lorsque vous travaillerez sur le code.

Dernière mise à jour : 14 juillet 2026
Projet : Tansa Formation
Prérequis : chapitres 1, 3 et 4 de la Partie I.

> **Objectifs du chapitre**
> À la fin de ce chapitre, le lecteur saura expliquer la frontière entre Bitcoin Core et Tansa, distinguer données strictes et projections reconstructibles, suivre le traitement complet d'un bloc, comprendre checkpoints, audits, idempotence et rejeu, puis diriger l'IA sans lui abandonner les décisions d'architecture.

Note de version

Ce chapitre décrit l'architecture consolidée de Layer1. Les responsabilités et les invariants sont durables ; certains noms de jobs et certains ordres d'exécution ont évolué pendant les refactorisations. Le code actif doit être relu avant d'affirmer le déroulement exact d'une version déployée.

1. La question de départ

Bitcoin Core sait valider et exposer les blocs du réseau Bitcoin. Pourquoi Tansa ne pourrait-il pas simplement interroger le nœud chaque fois qu'il a besoin d'une information ?

Parce qu'une application d'analyse doit répondre à d'autres besoins :

- retrouver rapidement les outputs créés et dépensés ;
- fournir des données adaptées aux modules suivants ;
- reprendre proprement après une interruption ;
- démontrer qu'un bloc a été traité intégralement ;
- distinguer une donnée indispensable d'une projection reconstruisible ;
- empêcher une erreur silencieuse de contaminer toutes les analyses.

Layer1 est la frontière entre Bitcoin Core et le système analytique de Tansa.

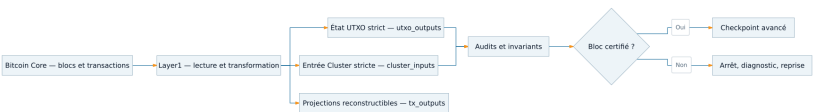
Mission fondamentale

Layer1 transforme les données validées par Bitcoin Core en faits structurés, contrôlés et certifiés pour le reste de Tansa.

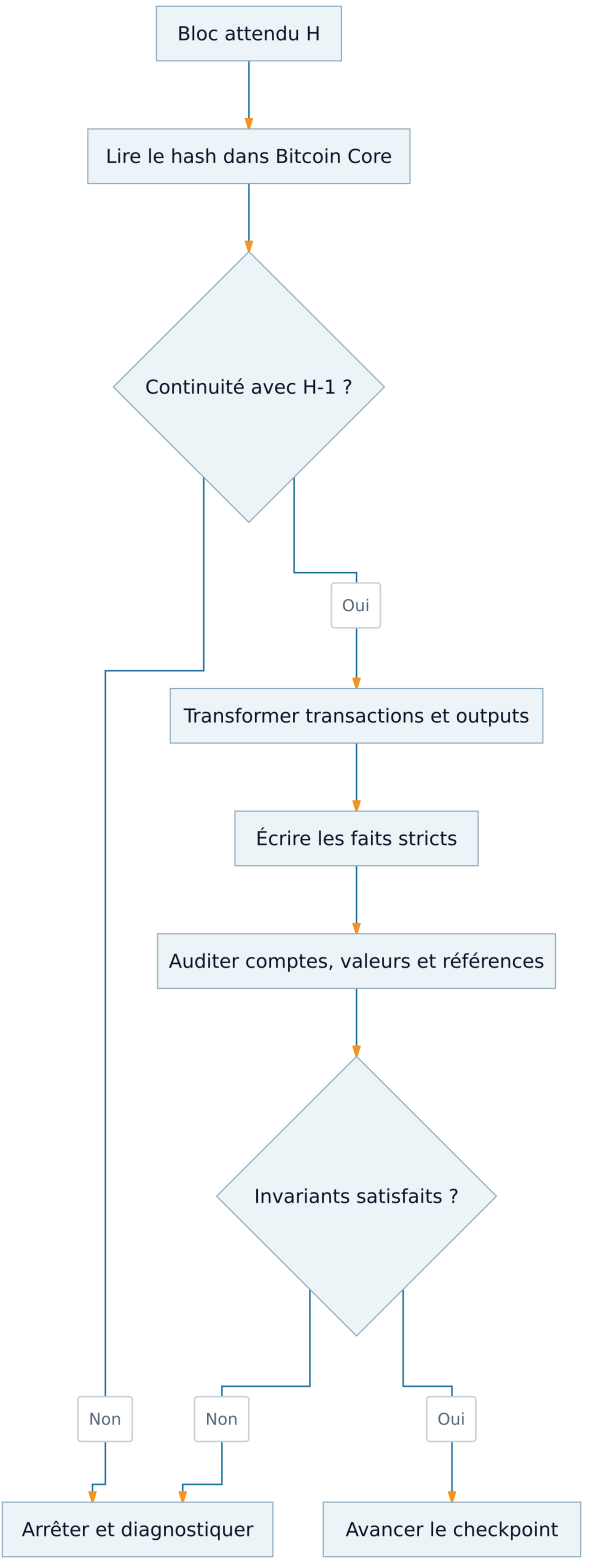
Le sas entre deux mondes

Bitcoin Core ressemble à l'autorité qui contrôle les passeports. Layer1 est le sas qui reçoit les voyageurs déjà contrôlés, vérifie que le groupe est complet, range leurs informations dans le format attendu par Tansa et n'ouvre la porte suivante qu'après ses propres audits.

2. La représentation mentale essentielle



Layer1 n'est pas seulement un importateur. C'est un sas de confiance.


Bloc attendu H
Lire le hash dans Bitcoin Core
Continuité avec H-1 ?
Oui
Transformer transactions et outputs
Écrire les faits stricts
Auditer comptes, valeurs et références
Invariants satisfaits ?
Non
Non
Oui
Arrêter et diagnostiquer
Avancer le checkpoint

3. La mission de Layer1

La mission peut être formulée comme un contrat.

Entrée

- une hauteur de bloc attendue ;
- le hash correspondant ;
- le bloc et ses transactions obtenus auprès de Bitcoin Core ;
- l'état strict déjà certifié au bloc précédent.

Sorties strictes

- les nouveaux outputs encore dépensables dans `utxo_outputs` ;
- la suppression ou le marquage cohérent des outputs dépensés ;
- les informations de dépense nécessaires à Cluster dans `cluster_inputs` ;
- les preuves d'audit du bloc ;
- un checkpoint avancé uniquement après validation.

Sorties reconstructibles

- l'historique complet ou enrichi des outputs dans `tx_outputs` ;
- les tables de projection et autres vues destinées à la consultation.

Hors de son périmètre

Layer1 ne doit pas :

- attribuer une identité à une adresse ;
- décider que plusieurs adresses appartiennent au même acteur ;
- qualifier un comportement ;
- produire un label métier ;
- transformer une observation en conclusion humaine.

Il transporte des faits Bitcoin. Les interprétations appartiennent aux modules suivants.

> **Terminologie à connaître**
>
> **Fait strict** — Donnée qui doit être correcte avant la progression du pipeline.
> **Projection reconstructible** — Vue utile que l'on peut régénérer depuis les sources certifiées.
> **Checkpoint** — Hauteur et contexte jusqu'auxquels un module affirme avoir satisfait son contrat.
> **Invariant** — Propriété qui doit rester vraie, par exemple l'égalité entre un comptage attendu et un comptage persisté.

4. Fait strict et projection reconstructible

Cette distinction est au cœur de l'architecture.

Un **fait strict** doit être correct avant que le pipeline puisse progresser. S'il manque, un consommateur en aval pourrait construire une

conclusion fausse ou incomplète.

Une **projection reconstructible** améliore la consultation ou les performances, mais peut être régénérée à partir des sources conservées. Son retard ne doit pas nécessairement bloquer le traitement strict.

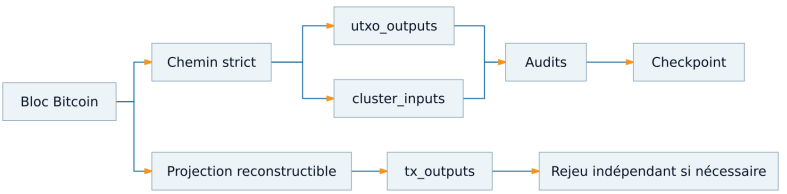
Donnée	Nature	Effet en cas d'échec
État des UTXO vivants	Strict	Le bloc ne doit pas être certifié
Dépenses transmises à Cluster	Strict	Le pipeline aval ne doit pas progresser
Comptages et valeur audités	Strict	Le checkpoint reste bloqué
Historique complet dans `tx_outputs`	Reconstructible	Projection à reprendre sans invalider les faits stricts déjà certifiés

Une première architecture peut placer trop de travail dans le chemin strict. Elle paraît rassurante, mais elle augmente la durée du verrou logique, le volume d'écritures et le nombre de causes capables d'arrêter tout le pipeline.

L'évolution de Layer1 V5 applique une règle plus précise :

> **Le chemin critique ne contient que ce qui est nécessaire à la vérité, à l'audit et au contrat avec le module suivant.**

`tx_outputs` a donc été sorti du chemin strict. La table demeure utile, mais sa projection peut travailler de manière asynchrone.



5. Les représentations principales des données

`utxo_outputs`

Cette table représente l'état vivant des outputs non dépensés connu par Tansa.

Elle répond à la question :

> Quels outputs peuvent encore être référencés comme entrées par une transaction future ?

Lorsqu'un bloc crée un output, celui-ci entre dans l'état UTXO. Lorsqu'un input le dépense, il en sort. La table est donc un état courant, pas nécessairement une archive complète de tout ce qui a existé.

`cluster_inputs`

Cette table constitue le contrat spécialisé entre Layer1 et Cluster. Elle transmet les dépenses dont Cluster a besoin pour appliquer ses règles de regroupement.

Cluster ne devrait pas relire tout Bitcoin Core ni parcourir l'intégralité de `tx_outputs` pour redécouvrir ces relations. Layer1 produit une entrée ciblée, contrôlée et rejouable.

`tx_outputs`

Cette table porte une projection historique complète des outputs. Elle est utile pour les recherches, les analyses et certaines restitutions, mais elle n'est plus une condition de certification du chemin strict.

`block_buffers`

Cette table suit le cycle de traitement des blocs mis en attente. Des états comme `enqueued`, `processing` et `processed` permettent de savoir si un bloc attend, est en cours ou a terminé son traitement prévu.

Ces états doivent être interprétés avec les checkpoints et les audits. Un simple statut `processed` ne remplace pas une preuve de cohérence.

Tables d'audit et de projection

Des enregistrements comme `AuditBlock`, `RecentBlockCadence` et `Layer1AuditRun` conservent les résultats de contrôle ou les mesures d'exploitation.

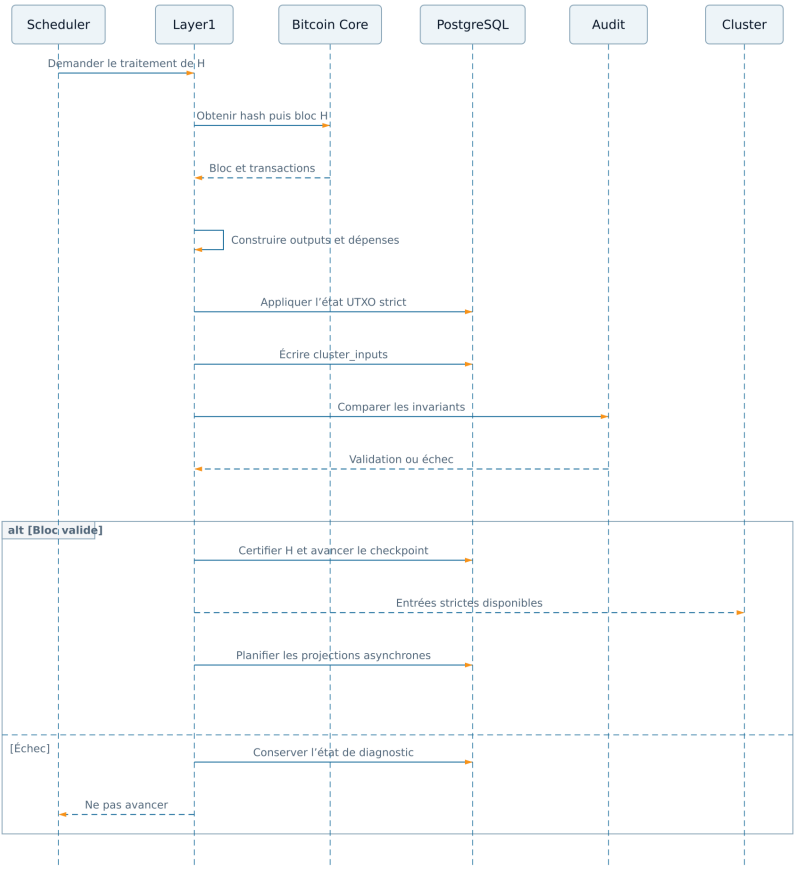
La projection asynchrone peut être suivie par `layer1_tx_output_projection_blocks`, avec des états tels que :

- `pending` ;
- `processing` ;
- `projected` ;
- `failed` ;
- `stale`.

Le nom exact des modèles Rails doit être vérifié dans la version courante du code. Le principe durable est de rendre l'avancement et l'échec observables bloc par bloc.

6. Le voyage d'un bloc

Prenons un bloc de hauteur `H`.



Le déroulement conceptuel est le suivant :

1. le scheduler sélectionne la prochaine hauteur admissible ;
2. Layer1 vérifie la continuité avec le checkpoint précédent ;
3. le hash et le bloc sont lus depuis Bitcoin Core ;
4. les transactions sont parcourues ;
5. les outputs créés sont préparés ;
6. les outputs référencés par les inputs sont résolus ;
7. l'état UTXO et `cluster_inputs` sont mis à jour ;
8. les invariants du bloc sont audités ;
9. le checkpoint avance seulement si la partie stricte est complète ;
10. les projections reconstructibles peuvent continuer séparément.

L'ordre technique précis peut être découpé entre plusieurs jobs, buffers et transactions SQL. Ce qui importe est le maintien de ce contrat logique.

7. Hauteur, hash et continuité

Une hauteur seule ne suffit pas à identifier durablement un bloc. Lors d'une réorganisation de chaîne, un autre bloc peut occuper la même hauteur.

Layer1 doit donc associer au minimum :

- la hauteur ;
- le hash du bloc ;
- le hash du bloc précédent ;
- le statut de traitement ;
- le résultat des contrôles.

Avant d'enchaîner `H` après `H - 1`, le système doit vérifier que la relation entre les hashes est cohérente avec la chaîne retenue par Bitcoin Core.

La gestion complète des réorganisations mérite un chapitre dédié. Ici, retenons que le checkpoint ne signifie pas seulement « nous avons vu cette hauteur », mais « nous avons certifié ce bloc précis dans une chaîne précise ».

8. Construire les outputs

Pour chaque transaction, Layer1 parcourt ses outputs et extrait les faits utiles :

- le `txid` ;
- l'index de l'output, souvent appelé `vout` ;
- la valeur en satoshis ;
- la condition de dépense ou sa représentation normalisée ;
- les informations de bloc ;
- les métadonnées strictement dérivables nécessaires au pipeline.

L'identifiant logique d'un output est l'outpoint :

```
txid:vout
```

L'association doit être unique. Rejouer le même bloc ne doit pas créer deux fois le même output.

Une transaction coinbase ne dépense pas un output antérieur. Elle crée néanmoins des outputs qui rejoignent l'état UTXO, sous réserve des règles de maturité appliquées lors d'une future dépense.

9. Résoudre les outputs dépensés

Chaque input ordinaire référence un outpoint antérieur. Layer1 doit retrouver cet output pour déterminer notamment sa valeur et la condition qui avait été verrouillée.

Conceptuellement :

```
input du bloc H
    └── référence txid:vout
            └── recherche dans l'état connu
                    ├── trouvé : produire le fait de dépense
                    └── absent : arrêter ou orienter vers une reprise contrôlée
```

Une référence absente ne doit pas être silencieusement ignorée. Elle peut signaler :

- un état UTXO incomplet ;
- un mauvais ordre de traitement ;
- une projection confondue avec la source stricte ;
- un défaut de reprise ;
- une incohérence de chaîne ;

- un bug d'implémentation.

Des composants rencontrés dans l'évolution de Tansa, comme `SpentOutputFlusherV2` ou `SpentOutputResolveJob`, illustrent la séparation entre accumulation, résolution et écriture. Ces noms documentent l'histoire du système, pas une garantie sur la version actuellement déployée.

10. Pourquoi utiliser des buffers Redis ?

Écrire une ligne PostgreSQL après chaque élément isolé provoquerait beaucoup d'allers-retours et de petites transactions. Layer1 peut préparer des groupes d'outputs et de dépenses dans des buffers Redis avant une écriture en lot.

Les buffers permettent :

- de regrouper les écritures ;
- de découpler temporairement lecture et persistance ;
- d'absorber certaines variations de cadence ;
- de mesurer le travail en attente ;
- d'appliquer une contre-pression lorsque PostgreSQL ne suit plus.

Mais Redis ne devient pas pour autant la source de vérité durable. Un buffer peut être perdu, dupliqué ou rejoué selon les incidents. La conception doit donc préciser :

- comment un lot est identifié ;
- quand il est considéré comme persisté ;
- comment il est rejoué ;
- comment un doublon est neutralisé ;
- quand le bloc peut être certifié.

11. `cluster_inputs` : un contrat, pas une copie arbitraire

Le module Cluster s'intéresse particulièrement aux inputs d'une même transaction, car certaines heuristiques utilisent leur coprésence.

Layer1 connaît déjà les dépenses pendant son traitement. Lui demander de produire `cluster_inputs` évite de refaire plus tard une lecture coûteuse et potentiellement divergente.

Cette table doit contenir uniquement les faits nécessaires au contrat. Elle ne doit pas contenir les conclusions de Cluster.

Layer1 peut affirmer	Layer1 ne doit pas affirmer
Cet input dépense cet outpoint	Ces deux adresses appartiennent certainement à la même personne
Ces inputs apparaissent dans la même transaction	Cette transaction provient d'un exchange
La valeur dépensée est celle de l'output référencé	Ce flux représente une vente

Layer1 peut affirmer	Layer1 ne doit pas affirmer
Le script observé possède telle forme	L'utilisateur avait telle intention

Cette frontière protège l'auditabilité : on peut modifier une heuristique de clustering sans réécrire les faits Bitcoin.

12. La projection asynchrone de `tx_outputs`

`tx_outputs` conserve une utilité importante, mais son remplissage complet augmente fortement le volume d'écritures. Sur une base PostgreSQL installée sur disque dur, ce travail peut entrer en concurrence avec l'état UTXO strict et ralentir toute la chaîne.

La solution architecturale consiste à :

1. certifier le minimum strict ;
2. enregistrer qu'une projection reste à produire ;
3. traiter cette projection dans une file Sidekiq distincte ;
4. suivre son statut par bloc ;
5. réessayer ou reconstruire sans reculer le checkpoint strict, sauf si une règle explicite l'exige.

La séparation ne signifie pas que l'historique est négligé. Elle signifie que sa disponibilité et la vérité stricte sont deux contrats différents.

13. Les audits : prouver avant d'avancer

Une exécution sans exception ne constitue pas une preuve suffisante. Un job peut terminer tout en ayant ignoré des lignes ou produit une somme erronée.

Pour chaque bloc, Layer1 peut comparer :

- le hash attendu et le hash réellement lu ;
- le nombre de transactions ;
- le nombre d'outputs ;
- la somme des valeurs créées ;
- le nombre de dépenses ordinaires ;
- le nombre d'éléments écrits dans les sorties strictes ;
- les éventuels doublons ou références non résolues.

Un enregistrement `AuditBlock` peut notamment conserver `block_hash`, `tx_count`, `outputs_count` et `outputs_value`.

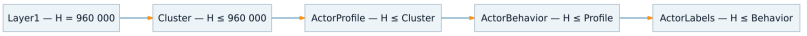
La règle essentielle est :

```
travail terminé + invariants validés = bloc certifiable
```

et non :

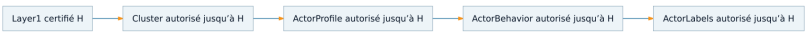
```
job terminé = bloc nécessairement correct
```

14. La chaîne de checkpoints



> **Une chaîne ne dépasse jamais son maillon amont**
>
> Un module aval peut être en retard. Il ne doit jamais certifier une hauteur supérieure à la source dont il dépend.

Un checkpoint représente la dernière progression certifiée d'un module.



Un module aval ne doit pas dépasser les faits certifiés du module dont il dépend. Cette chaîne rend les dépendances explicites et empêche une interface de présenter comme définitive une analyse calculée sur un socle incomplet.

Layer1 ne doit donc avancer son checkpoint qu'après :

- persistance de l'état UTXO strict ;
- persistance du contrat `cluster_inputs` ;
- réussite des audits obligatoires ;
- absence d'erreur bloquante connue.

15. Idempotence et rejeu

Un job asynchrone peut être exécuté plusieurs fois : perte d'accusé de réception, arrêt du processus, délai dépassé ou relance manuelle.

Layer1 doit être **idempotent** : rejouer la même unité de travail doit produire le même état final sans doubler les effets.

Les mécanismes possibles comprennent :

- contraintes uniques sur les identifiants naturels ;
- insertions avec gestion explicite des conflits ;
- statuts de lots ;
- transactions SQL limitées et cohérentes ;
- comparaison du hash avant reprise ;
- nettoyage ciblé d'une projection incomplète ;
- séparation entre préparation et certification.

L'idempotence ne doit pas être supposée parce que Sidekiq réessaie automatiquement. Elle doit être conçue et testée au niveau des données.

16. Pannes et reprises

Arrêt avant toute écriture stricte

Le bloc reste en attente ou revient dans une file. Aucun checkpoint ne doit avancer.

Arrêt pendant une écriture atomique

La transaction PostgreSQL doit être annulée. Le bloc peut être rejoué.

Arrêt après une partie des lots

Le système doit reconnaître les lots déjà persistés et poursuivre ou reconstruire sans duplication.

Audit en échec

Le bloc reste non certifié. Les valeurs observées doivent être conservées pour le diagnostic.

Projection `tx_outputs` en échec

Le statut passe à `failed` ou à un état équivalent. La projection est relancée séparément. Le chemin strict déjà certifié reste valable si ses propres invariants sont satisfaits.

Bloc devenu obsolète

Une réorganisation peut rendre un travail `stale`. La hauteur, le hash et la dépendance au bloc précédent permettent de déterminer ce qui doit être invalidé et rejoué.

17. Le rôle du scheduler et de PipelineController

Sidekiq sait exécuter des jobs, mais il ne connaît pas naturellement toutes les dépendances métier de Tansa.

Le scheduler ou `PipelineController` doit notamment décider :

- quel module a le droit de travailler ;
- quelle hauteur est admissible ;
- quelle file a la priorité ;
- si PostgreSQL ou Redis montre des signes de saturation ;
- si un rattrapage doit continuer ;
- si une projection asynchrone doit céder la place au strict ;
- si un module aval doit attendre.

Des files distinctes comme `layer1_strict`, `tx_outputs_async`, `cluster_strict` et `scheduler` expriment des responsabilités et des priorités différentes. Les noms précis peuvent changer ; le principe reste d'empêcher des travaux indépendants en apparence de se battre pour la même ressource critique.

18. Mesurer la progression réelle

Une file vide n'indique pas toujours que le système est à jour. Un worker occupé n'indique pas toujours qu'il rattrape son retard.

Le verdict principal doit provenir de la variation du retard observé :

```
retard = hauteur de la chaîne Bitcoin - checkpoint certifié Layer1
```

- si le retard diminue, Layer1 rattrape ;
- s'il reste stable près de zéro, Layer1 suit la chaîne ;
- s'il augmente, la capacité moyenne est insuffisante ou un blocage existe.

La cadence récente, éventuellement enregistrée par `RecentBlockCadence`, complète ce verdict. Elle ne le remplace pas.

Une interface peut afficher des états compréhensibles tels que :

- « À jour — en attente du prochain bloc » ;
- « Traitement en cours » ;
- « Rattrapage » ;
- « Attention » ;
- « Bloqué ».

Ces libellés doivent être calculés à partir de mesures explicites, pas d'une impression visuelle.

19. PostgreSQL comme goulot d'étranglement

Pendant le développement réel de Tansa, la limite principale de Layer1 n'a pas toujours été le calcul Ruby ni Bitcoin Core. PostgreSQL et les entrées-sorties disque ont constitué le goulot d'étranglement.

Des attentes telles que `IO/DataFileRead` indiquent que la base attend des pages de données. Sur disque dur, les accès aléatoires sont particulièrement coûteux.

Plusieurs leviers ont été étudiés ou appliqués :

- regrouper les insertions ;
- retirer `tx_outputs` du chemin strict ;
- réduire les lectures inutiles ;
- donner la priorité aux files strictes ;
- limiter la concurrence entre modules ;
- adapter les index aux requêtes réelles ;
- augmenter `shared_buffers` de 4 Go à 8 Go dans l'environnement observé ;
- mesurer séparément chaque phase.

Des blocs autrefois traités en dix à quinze minutes sont descendus sous cinq minutes dans les observations du projet, avec certaines phases d'insertion mesurées autour de 114 à 156 secondes après réglage. Ces valeurs sont des résultats d'un environnement précis, pas des garanties universelles.

La leçon durable est plus importante que les chiffres :

Optimiser exige d'identifier la ressource réellement attendue, puis de modifier l'architecture ou l'accès aux données en conséquence.

20. Rails dans cette architecture

Rails ne se limite pas ici au modèle MVC d'une application web classique.

Élément Rails	Responsabilité possible dans Layer1
Modèles Active Record	Contraintes et accès aux états persistés
Services	Transformation d'un bloc, résolution et règles de certification
Jobs Sidekiq / Active Job	Exécution asynchrone et reprise
Migrations	Évolution contrôlée du schéma et des index
Validations de base	Protection locale, sans remplacer les contraintes SQL
Tests	Preuve des invariants, du rejeu et des cas d'échec
Contrôleurs et vues	Pilotage et observabilité, pas logique centrale du pipeline

Un modèle Active Record ne devrait pas contenir à lui seul tout le traitement d'un bloc. Des services spécialisés rendent les responsabilités plus lisibles et testables. Inversement, multiplier les services sans contrat clair ne crée pas automatiquement une bonne architecture.

Le concepteur assisté par IA doit pouvoir répondre à trois questions pour chaque classe proposée :

1. quelle responsabilité unique porte-t-elle ?
2. quelles données reçoit-elle et produit-elle ?
3. quelle preuve montre que son travail est correct ?

21. Vérité, auditabilité, traçabilité, performance

L'ordre de priorité de Tansa est :

1. vérité ;
2. auditabilité ;
3. traçabilité ;
4. performance.

La performance est indispensable pour suivre Bitcoin, mais elle ne justifie pas de supprimer une vérification nécessaire.

La bonne question n'est pas « comment aller plus vite à tout prix ? », mais :

> Quelles écritures doivent absolument bloquer la certification, et quelles écritures peuvent être déplacées sans affaiblir la preuve ?

C'est précisément cette réflexion qui permet de sortir une projection reconstructible du chemin critique tout en conservant les audits stricts.

22. Ce que l'IA peut faire — et ce que le développeur doit décider

L'IA peut aider à :

- cartographier les jobs et leurs dépendances ;
- proposer une migration ;
- générer une contrainte unique ;
- écrire des tests d'idempotence ;
- analyser un plan de requête ;
- comparer des mesures avant et après ;
- documenter un incident.

Le développeur ou architecte reste responsable de :

- définir ce qui constitue la vérité stricte ;
- choisir le contrat entre Layer1 et Cluster ;
- décider quand un checkpoint peut avancer ;
- évaluer le risque d'une reprise ;
- vérifier que les métriques prouvent réellement l'amélioration ;
- refuser une optimisation qui détruit l'auditabilité.

Une proposition de code élégante n'est pas une preuve. Elle devient acceptable lorsque les invariants, les tests et les observations réelles la confirment.

23. Tests et invariants indispensables

Une suite de tests Layer1 devrait couvrir au minimum :

- un bloc contenant une coinbase et des transactions ordinaires ;
- la création unique de chaque outpoint ;
- la disparition correcte d'un UTXO dépensé ;
- la création exacte des `cluster_inputs` attendus ;
- le rejeu du même bloc sans doublon ;
- une référence d'input introuvable ;
- un arrêt avant certification ;
- un audit de comptage en échec ;
- une projection historique en échec sans corruption du strict ;
- deux blocs consécutifs avec contrôle des hashes ;
- une hauteur identique associée à un hash différent ;
- l'impossibilité pour Cluster de dépasser le checkpoint Layer1.

Il faut combiner :

- tests unitaires des transformations ;
- tests d'intégration avec PostgreSQL et Redis ;

- tests de jobs et de rejeu ;
- audits sur des blocs réels connus ;
- mesures en environnement d'exploitation.

24. Exemple simplifié

Imaginons un bloc contenant deux transactions :

```
Transaction A — coinbase
  crée A:0 = 300 000 000 sat

Transaction B
  dépense X:1 = 80 000 sat
  crée B:0 = 50 000 sat
  crée B:1 = 29 000 sat
  frais       = 1 000 sat
```

Après traitement strict :

- `A:0`, `B:0` et `B:1` entrent dans `utxo_outputs` ;
- `X:1` quitte l'état UTXO ;
- la dépense de `X:1` produit une entrée exploitable dans `cluster_inputs` ;
- les comptages et sommes du bloc sont comparés aux données décodées ;
- le checkpoint peut avancer si toutes les autres transactions et tous les invariants concordent ;
- la projection de `A:0`, `B:0` et `B:1` dans `tx_outputs` peut être effectuée ensuite.

Layer1 ne conclut pas que `B:1` est forcément la monnaie rendue, même si cette hypothèse peut être examinée par un module analytique. Il enregistre d'abord ce qui est certain.

25. Erreurs fréquentes

Confondre Bitcoin Core et Layer1

Bitcoin Core valide Bitcoin. Layer1 organise les faits nécessaires à Tansa et prouve la cohérence de leur acquisition.

Confondre état vivant et historique

`utxo_outputs` répond à une question d'état courant. `tx_outputs` répond à une question d'historique projeté.

Faire relire Bitcoin Core à chaque module

Cela duplique le travail, brouille les responsabilités et peut produire des versions divergentes d'un même fait.

Avancer le checkpoint dès que le job se termine

La fin technique d'un job ne garantit ni les comptages ni la complétude.

Traiter Redis comme une vérité permanente

Redis facilite le flux et les lots. La certification repose sur un protocole de persistance et des audits explicites.

Optimiser sans mesurer

Augmenter le nombre de workers peut aggraver la contention PostgreSQL. Une file plus concurrente n'est pas nécessairement une file plus rapide.

Transformer une heuristique en fait Layer1

La présence de plusieurs inputs est un fait. Leur appartenance à une même entité est une inférence qui doit rester traçable et révisable.

26. Ce qui est stable et ce qui doit être revérifié

Architecture stable

- Layer1 est la source des faits stricts de Tansa ;
- `utxo_outputs` porte l'état UTXO vivant ;
- `cluster_inputs` est le contrat direct vers Cluster ;
- `tx_outputs` est une projection historique reconstructible ;
- les audits précèdent la certification ;
- les checkpoints ordonnent les modules ;
- le rejeu doit être idempotent ;
- la progression se juge par le retard certifié.

Implémentation à vérifier dans le code actif

- nom exact des jobs et services ;
- ordre exact des flushers et resolveurs ;
- contenu précis de chaque file Sidekiq ;
- transaction SQL couvrant chaque phase ;
- règles actuelles de réessai ;
- seuils d'alerte et de cadence ;
- prise en charge complète des réorganisations ;
- colonnes exactes des tables d'audit et de projection.

Cette séparation évite de figer dans la formation une photographie technique qui pourrait déjà avoir évolué.

27. Exercices de compréhension

Atelier guidé — raisonner comme l'architecte de Tansa

Voici une méthode simple pour aborder un exemple de ce chapitre :

1. Vérifier hauteur, hash et continuité.
2. Séparer le chemin strict des projections reconstructibles.
3. N'avancer le checkpoint qu'après les audits.

Mini-défi

Pourquoi un job terminé sans exception ne suffit-il pas à certifier un bloc ?
Prenez quelques secondes pour formuler une réponse. Une phrase précise vaut mieux qu'un long paragraphe rempli de suppositions.

Corrigé raisonné

L'absence d'exception prouve seulement que le programme a atteint sa fin. Les audits doivent encore comparer les quantités, valeurs, hashes et transitions attendues.
Le but n'est pas seulement d'obtenir la bonne réponse. Il est de séparer ce qui est observé, ce qui est calculé et ce qui reste une hypothèse.

Pause — trois idées à garder

- Layer1 protège les faits.
- Le checkpoint est une promesse certifiée.
- Une projection peut être rejouée si sa source demeure intacte.

Exercice 1 — Reformuler le contrat

Expliquez en trois phrases ce que Layer1 reçoit, ce qu'il garantit et ce qu'il refuse d'interpréter.

Exercice 2 — Classer les données

Classez chaque élément comme strict, reconstructible ou analytique :

- état des UTXO vivants ;
- historique complet des outputs ;
- association de deux adresses à un même acteur ;
- hash du bloc certifié ;
- projection destinée à une page de recherche.

Exercice 3 — Décider d'un checkpoint

Tous les UTXO et `cluster_inputs` sont persistés, les audits stricts réussissent, mais la projection `tx_outputs` est en échec. Le checkpoint Layer1 peut-il avancer selon l'architecture V5 ? Justifiez.

Exercice 4 — Diagnostiquer une lenteur

Le nombre de workers est doublé, mais le retard augmente et PostgreSQL passe davantage de temps en `IO/DataFileRead`. Proposez trois vérifications avant toute nouvelle modification.

Exercice 5 — Diriger l'IA

Rédigez une mission demandant à une IA d'ajouter un test de rejeu d'un bloc. La mission doit préciser l'invariant, les données initiales, le résultat attendu et ce que l'IA ne doit pas modifier.

Corrigé commenté

1. Reformuler le contrat. Layer1 reçoit un bloc validé par Bitcoin Core, son hash et l'état certifié précédent. Il garantit la production complète et auditée des faits stricts nécessaires à Tansa, notamment l'état UTXO et l'entrée de Cluster. Il refuse d'attribuer une identité, un comportement ou un label.

2. Classer les données. L'état des UTXO vivants est strict. L'historique complet des outputs et la projection de recherche sont reconstructibles. L'association de deux adresses à un acteur est analytique et appartient à Cluster. Le hash du bloc certifié fait partie du contrat strict.

3. Décider d'un checkpoint. Oui, dans l'architecture V5 décrite ici, le checkpoint strict peut avancer si les UTXO, `cluster_inputs` et tous les audits stricts sont validés. L'échec de `tx_outputs` doit être enregistré et rejoué, sans transformer cette projection asynchrone en condition cachée de certification.

4. Diagnostiquer une lenteur. Il faut mesurer les requêtes dominantes et leurs plans, le taux de lecture disque et l'efficacité du cache PostgreSQL, puis vérifier tailles de lots, index, concurrence et temps d'attente. Doubler les workers peut simplement augmenter la contention et les lectures aléatoires.
5. Diriger l'IA. Une mission correcte peut demander : « Ajoute un test rejouant deux fois le bloc H à partir du même état certifié. Vérifie que le nombre d'UTXO, leurs valeurs, les `cluster_inputs`, les audits et le checkpoint sont identiques après les deux passages, sans doublon. Utilise les fixtures du bloc H et de H−1. Ne modifie ni le schéma, ni les services de production, ni les règles de checkpoint ; propose séparément toute correction découverte. »

28. Synthèse

Layer1 est le premier niveau de confiance propre à Tansa.

Il :

- lit les blocs retenus par Bitcoin Core ;
- transforme transactions, outputs et dépenses en faits structurés ;
- maintient l'état UTXO strict ;
- fournit à Cluster une entrée spécialisée ;
- sépare le chemin critique des projections reconstructibles ;
- rend chaque étape observable et rejouable ;
- vérifie des invariants avant toute certification ;
- avance un checkpoint qui limite les modules en aval.

Leçon d'architecture

Un pipeline fiable ne progresse pas parce qu'un programme a fini de s'exécuter. Il progresse parce que des preuves définies à l'avance montrent que son contrat est satisfait.

29. Prochaine étape

Le prochain chapitre sera :

`11-tansa-formation-cluster.md`

Il répondra à la question suivante :

> Comment Tansa passe-t-il des dépenses certaines produites par Layer1 à des regroupements d'adresses utiles, sans présenter une heuristique comme une vérité absolue ?

Partie III — Programmer et architecturer avec l’IA

Chapitre 13 — Le nouveau métier de développeur avec l'IA

L'IA écrit du code ; le développeur construit un système fiable

Avant de commencer — une scène concrète

Une IA produit en quelques secondes une migration, un job et trois tests. Tout paraît cohérent. Pourtant, elle ignore peut-être qu'une table contient des millions de lignes et qu'un module temps réel partage le même disque.

Cette scène contient le problème que le chapitre va résoudre. Ne cherchez pas encore à mémoriser les noms de classes, de tables ou de commandes. Commencez par comprendre **la responsabilité** : quelle question faut-il trancher, avec quelles données, et quelle conclusion serait excessive ?

> **L'image mentale du chapitre**
>
> L'IA est une machine-outil extrêmement rapide. Le développeur reste l'architecte qui choisit le plan, les tolérances, les contrôles et le moment d'arrêter la machine.

La carte du chapitre en une ligne

comprendre le besoin → inspecter l'existant → définir la mission → modifier peu → tester → lire le diff → mesurer

Lisez cette chaîne une première fois sans vous arrêter. À la fin du chapitre, vous pourrez revenir ici et expliquer chaque flèche avec vos propres mots.

> **Votre droit de ne pas tout retenir**
>
> Pour une première lecture, retenez le mouvement général et les limites du composant. Les détails d'implémentation servent à montrer que l'architecture est réelle ; vous pourrez les relire lorsque vous travaillerez sur le code.

> **Objectifs du chapitre**
>
> À la fin de ce chapitre, le lecteur saura :
>
> - expliquer ce que l'IA change réellement dans le développement logiciel ;
> - distinguer production de code et responsabilité

d'architecture ;
- transformer une intention en contrat vérifiable ;
- rédiger une mission de développement précise ;
- contrôler un diff, des tests, une migration et une mesure de performance ;
- découper un chantier en changements réversibles ;
- reconnaître les situations dans lesquelles l'IA doit être arrêtée ;
- organiser une collaboration durable entre concepteur et IA.

1. La question de départ

Si une intelligence artificielle peut produire un modèle Rails, une migration, un service, un job Sidekiq et les tests associés, faut-il encore apprendre à développer ?

Oui, mais la compétence centrale se déplace.

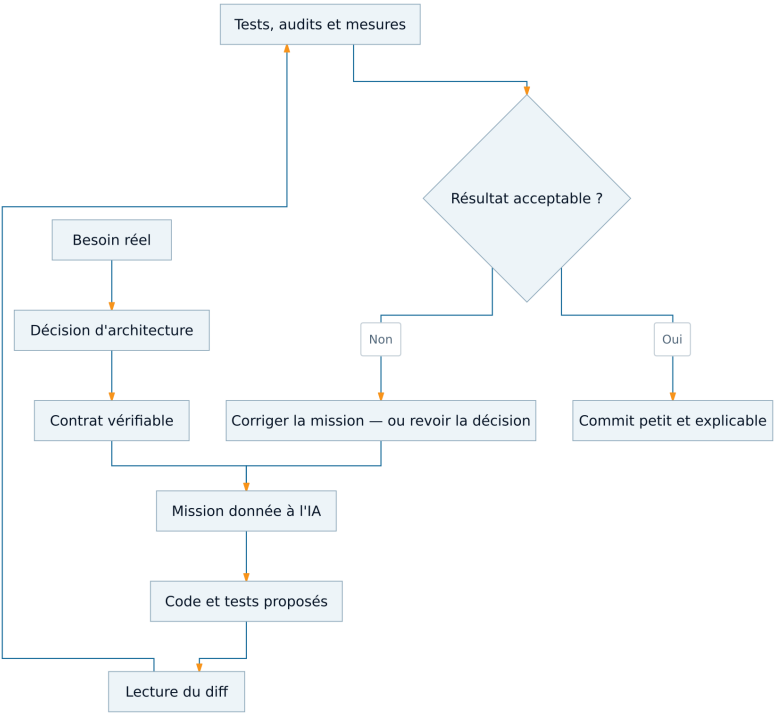
Écrire chaque ligne de mémoire devient moins important. Savoir ce que le système doit garantir, comment ses composants se partagent les responsabilités et quelles preuves accepter devient plus important.

Le développeur ne disparaît pas. Il devient davantage concepteur, architecte, contrôleur et responsable du résultat.

Idée essentielle

L'IA réduit le coût de production du code. Elle ne réduit pas automatiquement le coût d'une mauvaise décision, d'une donnée corrompue ou d'une architecture impossible à exploiter.

2. La représentation mentale essentielle



Cette boucle contient deux formes de travail différentes :

- **penser le système** : définir le besoin, le contrat, les frontières et les risques ;
- **produire une implémentation** : écrire le code, les tests et la documentation.

L'IA est particulièrement rapide dans la seconde. Le développeur reste responsable de la première et de la validation finale.

3. Ce que l'IA sait très bien faire

Une IA de programmation peut notamment :

- explorer une base de code ;
- retrouver les fichiers liés à un comportement ;
- produire une migration Rails ;
- écrire un modèle et ses validations ;
- extraire une logique dans un service ;
- construire un job Sidekiq ;
- proposer des tests ciblés ;
- expliquer un plan d'exécution PostgreSQL ;
- comparer plusieurs implémentations ;
- documenter une décision ;
- effectuer des transformations répétitives ;

- repérer certaines incohérences.

Sa vitesse change l'échelle des projets accessibles à une petite équipe. Une personne capable de diriger correctement l'outil peut construire une application qui aurait autrefois nécessité plusieurs spécialistes.

> **Une équipe extrêmement rapide**
>
> Imaginez une équipe qui connaît beaucoup de techniques, travaille sans fatigue et peut essayer plusieurs solutions en quelques minutes. Cette équipe reste néanmoins nouvelle dans votre entreprise : elle ne connaît pas spontanément vos priorités, vos incidents passés ni les promesses faites aux utilisateurs. Il faut lui transmettre le contexte et contrôler son travail.

4. Ce que l'IA ne peut pas décider seule

L'IA ne possède pas naturellement l'autorité nécessaire pour décider :

- quelle donnée constitue la vérité du produit ;
- quel risque est acceptable ;
- si une heuristique peut être présentée comme un fait ;
- si une migration peut bloquer une table de plusieurs gigaoctets ;
- si une optimisation affaiblit un audit ;
- si un changement respecte la promesse faite au client ;
- si un incident justifie l'arrêt du pipeline ;
- quel compromis doit être assumé par le projet.

Elle peut proposer. Elle peut argumenter. Elle peut parfois détecter un danger oublié. Mais le droit de modifier un système doit rester lié à une responsabilité humaine claire.

> **Une réponse convaincante n'est pas une preuve**
>
> Le texte d'une IA peut être fluide, précis et faux. Une explication ne remplace ni la lecture du code actif, ni un test, ni une contrainte SQL, ni une mesure obtenue sur le système réel.

5. Le développeur devient gardien des contrats

Un contrat logiciel décrit ce qu'un composant reçoit, ce qu'il produit, ce qu'il garantit et ce qu'il refuse de faire.

Pour Layer1, un contrat simplifié pourrait être :

```
Entrée : un bloc Bitcoin validé et l'état certifié précédent.
Sortie : les faits UTXO et les dépenses nécessaires à Cluster.
Garantie : le checkpoint avance seulement après les audits stricts.
Limite : Layer1 ne regroupe pas les adresses et n'attribue aucune identité.
```

Cette formulation guide le code. Elle indique aussi ce qu'un test doit prouver et ce qu'une mission donnée à l'IA ne doit pas modifier.

Terminologie à connaître

Contrat — Engagement vérifiable d'un composant envers ses consommateurs.
Invariant — Propriété qui doit rester vraie avant et après une opération.
Précondition — État qui doit être satisfait avant l'exécution.
Postcondition — État attendu après une exécution réussie.
Effet de bord — Modification observable en dehors de la valeur directement retournée : écriture SQL, message Redis, fichier ou job planifié.

6. De l'idée au changement vérifiable

Une idée comme « accélérer Layer1 » est trop vague. Elle ne dit pas ce qui est lent, ce qui doit rester strict ni comment reconnaître une amélioration.

Le travail de conception consiste à transformer cette idée en une suite de questions :

1. Quelle opération consomme le temps ?
2. Quelle mesure le démontre ?
3. Cette opération appartient-elle au chemin strict ?
4. Peut-elle devenir reconstructible ?
5. Quels consommateurs dépendent de son résultat ?
6. Quels invariants ne doivent jamais être affaiblis ?
7. Quel changement minimal permet de tester l'hypothèse ?
8. Comment revenir en arrière ?

L'IA peut aider à répondre à ces questions. Elle ne doit pas les supprimer en passant directement de l'idée au code.

7. La mission : unité de collaboration avec l'IA

Une bonne mission contient au minimum :

- **le résultat attendu** ;
- **le contexte utile** ;
- **le périmètre autorisé** ;
- **les fichiers ou contrats interdits de modification** ;
- **les invariants** ;
- **les tests à exécuter** ;
- **les critères d'acceptation** ;
- **la forme du compte rendu**.

Exemple :

```
Mission : rendre la projection tx_outputs asynchrone.

Résultat attendu :
- le checkpoint strict ne dépend plus de tx_outputs ;
- la projection peut être rejouée par hauteur ;
```

```
- aucun doublon n'est produit.

À préserver :
- utxo_outputs et cluster_inputs restent stricts ;
- les audits Layer1 ne sont pas affaiblis ;
- le contrat public de Cluster ne change pas.

Méthode :
1. analyser le code actif et présenter le chemin d'exécution ;
2. proposer le plus petit plan ;
3. attendre la validation du plan ;
4. implémenter avec tests ciblés ;
5. fournir le diff, les tests et les risques résiduels.

Interdictions :
- ne pas modifier les migrations existantes ;
- ne pas supprimer de test ;
- ne pas changer les seuils d'audit.
```

> **Une mission n'est pas seulement un prompt**
>
> Le prompt demande du texte. La mission organise une responsabilité : elle définit ce qui peut changer, ce qui doit rester vrai et les preuves nécessaires pour accepter le résultat.

8. Commencer par l'analyse de l'existant

Une application vivante contient toujours plus de réalité que sa documentation :

- des migrations appliquées mais absentes d'un schéma suivi ;
- un job encore présent mais plus planifié ;
- une file Sidekiq renommée ;
- un ancien service toujours appelé par un chemin secondaire ;
- une contrainte seulement implicite ;
- un test qui décrit mieux le contrat que le commentaire.

Avant d'écrire, l'IA doit donc examiner :

1. les points d'entrée ;
2. les appels entre services ;
3. les modèles et contraintes SQL ;
4. les jobs et leurs files ;
5. les tests existants ;
6. l'état Git ;
7. les instructions du projet ;
8. les métriques disponibles.

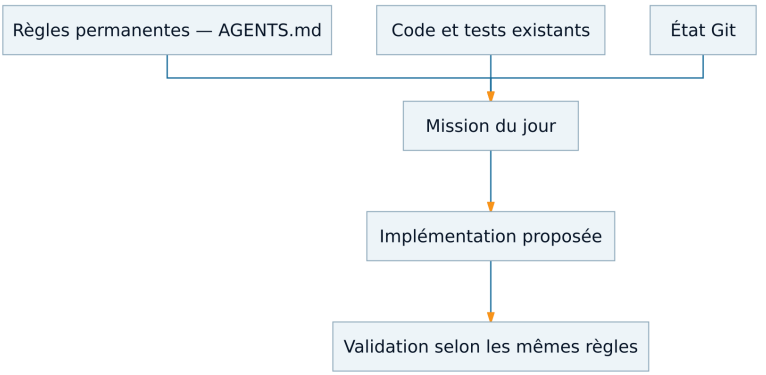
L'analyse doit produire une carte du comportement actuel, les zones d'incertitude et un plan limité.

9. Les instructions persistantes du projet

Répéter toutes les règles dans chaque mission est inefficace. Le projet peut conserver des instructions permanentes dans un fichier comme `AGENTS.md`.

Pour Tansa, ce document peut rappeler :

- vérité des données avant performance ;
- auditabilité avant raccourci ;
- séparation strict/reconstructible ;
- aucune suppression silencieuse de test ;
- petits commits cohérents ;
- tests ciblés proportionnés au risque ;
- préservation des changements non liés ;
- analyse obligatoire avant un refactoring.



Ces instructions ne remplacent pas la mission. Elles constituent le cadre commun dans lequel chaque mission est exécutée.

10. Travailler par petits changements

Une IA peut modifier cinquante fichiers en quelques secondes. Cette capacité est parfois utile, mais elle augmente le coût de vérification.

Un changement réduit offre plusieurs avantages :

- le diff reste lisible ;
- la cause d'une régression est plus facile à isoler ;
- le retour arrière est possible ;
- les tests peuvent viser un contrat précis ;
- l'historique Git raconte une décision ;
- une erreur de compréhension contamine moins de composants.

Une refactorisation importante peut être divisée :

1. introduire une table ou une abstraction sans changer le comportement ;
2. écrire les tests du nouveau contrat ;
3. brancher un premier producteur ;
4. migrer les consommateurs ;
5. observer les résultats ;
6. retirer l'ancien chemin dans un commit distinct.

Gros diff, petite compréhension

Un changement volumineux peut donner l'impression que le chantier est presque terminé alors que personne ne peut expliquer précisément toutes ses conséquences. La vitesse d'écriture ne doit pas dépasser durablement la capacité de vérification.

11. Lire un diff produit par l'IA

Le diff est le véritable produit intermédiaire de la collaboration. Il faut y chercher :

- les fichiers modifiés sans rapport avec la mission ;
- les suppressions inattendues ;
- les changements de contrat public ;
- les callbacks ou effets de bord ajoutés ;
- les requêtes exécutées dans une boucle ;
- les valeurs flottantes utilisées pour des satoshis ;
- les erreurs absorbées sans trace ;
- les migrations non réversibles ;
- les tests modifiés pour accepter un comportement plus faible ;
- les noms trompeurs.

Une bonne question n'est pas seulement « est-ce que le code fonctionne ? », mais « quelle nouvelle propriété ce diff introduit-il et comment est-elle prouvée ? »

12. Les tests ne servent pas à rassurer

Un test vert ne prouve que ce qu'il vérifie réellement.

Si un test affirme seulement qu'un job ne lève pas d'exception, il ne prouve pas :

- que toutes les lignes attendues ont été écrites ;
- qu'aucun doublon n'existe ;
- que le checkpoint est correct ;
- que le rejeu produit le même état ;
- que deux jobs concurrents ne se contredisent pas ;
- que la performance reste acceptable.

Les tests doivent dériver des contrats et des invariants.

Risque	Preuve adaptée
Ligne manquante	Comptage et contenu attendus
Doublon au rejeu	Deux exécutions, même état final
Course concurrente	Exécutions concurrentes contrôlées
Mauvais checkpoint	Sources incompatibles simulées
Régression SQL	Plan ou mesure comparée
Contrat aval cassé	Test d'intégration du consommateur

Trois niveaux de test

Test unitaire — Vérifie une règle locale avec peu de dépendances.
Test d'intégration — Vérifie la collaboration entre plusieurs composants réels.
Test de système — Vérifie un parcours complet proche de l'usage réel.

13. Les migrations exigent une vigilance particulière

Une migration change les données et parfois la disponibilité du système. Son risque dépend du volume réel, pas seulement de la simplicité du code Ruby.

Avant d'accepter une migration proposée par l'IA, il faut examiner :

- la taille de la table ;
- la durée probable du verrou ;
- la construction de l'index ;
- la présence d'une valeur par défaut ;
- le remplissage des anciennes lignes ;
- la compatibilité entre ancien et nouveau code ;
- la réversibilité ;
- la stratégie de déploiement, réservée au Tome 2 ;
- la cohérence entre migrations, `schema.rb` et base réellement appliquée.

Une migration de quelques lignes peut représenter l'opération la plus dangereuse d'un chantier.

14. Mesurer avant d'optimiser

Face à une lenteur, l'IA peut proposer rapidement : ajouter des workers, créer un index, augmenter la taille d'un lot ou mettre en cache.

Ces solutions peuvent aider ou aggraver le problème.

Dans Tansa, doubler les workers d'un traitement limité par les lectures PostgreSQL peut augmenter la concurrence sur le disque. L'action raisonnable commence par des mesures :

- durée totale ;
- requête dominante ;
- plan d'exécution ;
- blocs lus depuis le cache et le disque ;
- type d'attente PostgreSQL ;
- volume écrit ;
- contention ;
- progression réellement obtenue.

L’optimisation est une expérience

Une optimisation sérieuse possède une mesure initiale, une hypothèse, une modification limitée et une mesure finale. Sans comparaison, il ne s’agit que d’une préférence technique.

15. Déboguer : chercher la cause, pas seulement le message

Lorsqu’un module ne progresse plus, le premier message visible n’est pas toujours la cause.

Une méthode de diagnostic peut suivre cette chaîne :

1. définir le comportement attendu ;
2. constater précisément l’écart ;
3. retrouver le dernier état certifié ;
4. examiner les logs du producteur et du consommateur ;
5. vérifier files, retries et jobs actifs ;
6. examiner les attentes PostgreSQL ;
7. reproduire le problème dans un périmètre réduit ;
8. formuler une hypothèse falsifiable ;
9. tester sans élargir prématurément le changement.

L’IA aide à corréler les indices. Le développeur doit empêcher qu’elle transforme la première hypothèse plausible en certitude.

16. Savoir arrêter l’IA

Il faut interrompre une mission lorsque :

- le périmètre commence à s’élargir sans justification ;
- l’IA veut modifier un contrat non autorisé ;
- elle propose de supprimer un test gênant ;
- elle ne distingue plus le code observé de son hypothèse ;
- une migration destructive apparaît ;
- des données utilisateur ou des secrets risquent d’être exposés ;
- les tests disponibles ne peuvent pas prouver le résultat ;
- l’état Git contient des changements qu’elle ne comprend pas ;
- une décision produit est nécessaire.

Arrêter n’est pas un échec. C’est une fonction normale de gouvernance.

L'autonomie a des limites explicites

Une IA peut être autonome dans un périmètre réversible et contrôlé. Elle ne doit pas recevoir implicitement l'autorité d'élargir la mission, de publier, de supprimer des données ou de modifier une promesse produit.

17. Quand l'IA et le développeur ne sont pas d'accord

L'IA peut révéler une faiblesse réelle dans la demande. Le développeur ne doit donc pas la traiter comme un simple exécutant silencieux.

Un désaccord utile doit être reformulé :

- quel invariant chaque solution protège-t-elle ?
- quel risque chacune introduit-elle ?
- quelles données manquent pour décider ?
- peut-on construire une expérience ou un mode shadow ?
- la décision est-elle technique, produit, juridique ou opérationnelle ?

Le développeur conserve la décision, mais il doit être capable de l'expliquer.

18. Répartition des responsabilités

Responsabilité	IA	Développeur-architecte
Explorer le code	Très efficace	Oriente le périmètre
Proposer plusieurs solutions	Très efficace	Choisit les critères
Écrire le code	Très efficace	Autorise et contrôle
Définir la vérité des données	Peut conseiller	Responsable
Accepter un risque	Ne doit pas décider seule	Responsable
Lire le diff	Peut pré-analyser	Validation finale
Écrire des tests	Très efficace	Définit ce qu'ils doivent prouver
Mesurer	Automatise	Interprète la mesure
Modifier le produit	Peut suggérer	Décision humaine
Rendre des comptes	Produit un rapport	Assume la décision

Cette répartition rassure le lecteur : il n'a pas besoin d'être le meilleur spécialiste de chaque syntaxe. Il doit devenir compétent dans la définition, le contrôle et l'arbitrage.

19. Les erreurs fréquentes

Demander directement une fonctionnalité complète

Sans analyse, l'IA peut construire une seconde architecture à côté de celle qui existe.

Confondre quantité de code et progression

Un gros diff peut augmenter le travail restant.

Accepter les tests générés sans lire leurs assertions

Un test peut reproduire l'implémentation au lieu de protéger le contrat.

Laisser l'IA corriger Git au hasard

Des fichiers supprimés, des migrations divergentes ou des conflits ne doivent pas être réglés par une commande destructive improvisée.

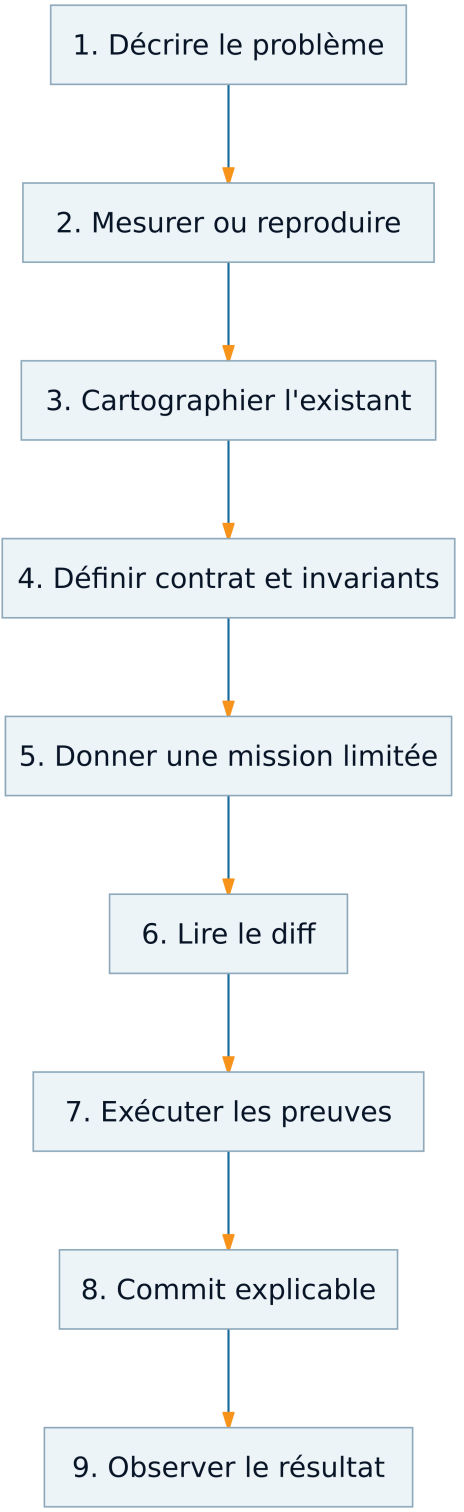
Optimiser sans mesure initiale

Le résultat devient impossible à attribuer au changement.

Présenter l'IA comme responsable

Un utilisateur ne peut pas demander des comptes à un modèle. L'équipe qui publie le logiciel reste responsable.

20. Une méthode quotidienne concrète



Cette méthode paraît plus lente qu'un ordre comme « fais-moi la fonctionnalité ». Sur une application complexe, elle évite surtout de perdre plusieurs jours à comprendre une modification trop vaste.

21. Exercices de compréhension

Atelier guidé — raisonner comme l'architecte de Tansa

Voici une méthode simple pour aborder un exemple de ce chapitre :

1. Écrire le résultat attendu avant le code.
2. Rendre les interdictions et invariants explicites.
3. Exiger une preuve proportionnée au risque.

Mini-défi

Un changement généré par l'IA passe tous les tests. Est-il automatiquement bon ?
Prenez quelques secondes pour formuler une réponse. Une phrase précise vaut mieux qu'un long paragraphe rempli de suppositions.

Corrigé raisonné

Non. Les tests ne couvrent que les promesses qu'ils expriment. Il faut encore examiner le périmètre, le diff, les risques de données, les performances et les hypothèses absentes.
Le but n'est pas seulement d'obtenir la bonne réponse. Il est de séparer ce qui est observé, ce qui est calculé et ce qui reste une hypothèse.

Pause — trois idées à garder

- L'IA propose ; le développeur engage le système.
- La mission vaut mieux qu'un prompt vague.
- Un diff compris est plus important qu'un volume de code produit.

Exercice 1 — Code ou architecture ?

Classez les tâches suivantes : produire une migration, décider si `tx_outputs` appartient au chemin strict, écrire un test de rejeu, choisir si un faux positif Cluster est acceptable, renommer une méthode.

Exercice 2 — Transformer une demande

La demande « rends ActorProfile plus rapide » est-elle suffisante ? Transformez-la en cinq questions vérifiables.

Exercice 3 — Contrôler un test

Un test exécute deux fois un job et vérifie seulement qu'aucune exception n'est levée. Pourquoi ne prouve-t-il pas l'idempotence ?

Exercice 4 — Lire une proposition de l'IA

L'IA propose de doubler les workers parce qu'un job est lent. Quelles données faut-il obtenir avant de décider ?

Exercice 5 — Écrire une mission

Rédigez une mission demandant d'ajouter un état `stale` à une projection Rails sans modifier les contrats stricts.

Corrigé commenté

1. Code ou architecture. Produire une migration, écrire un test et renommer une méthode sont principalement des tâches d'implémentation, même si elles nécessitent un contrôle. Décider si `tx_outputs` est strict et accepter ou non le risque d'un faux positif sont des décisions d'architecture et de produit. L'IA peut éclairer les cinq tâches ; elle ne doit pas trancher seule les deux dernières.

2. Transformer une demande. Il faut demander : quelle requête ou quel cluster consomme le temps ? Quelle mesure initiale le démontre ? Quelles sources et quels checkpoints doivent rester identiques ? Quelle modification minimale peut être essayée en shadow ? Quels indicateurs permettront de comparer débit, erreurs, lectures disque et fraîcheur avant/après ?

3. Contrôler un test. L'absence d'exception ne vérifie ni les doublons, ni les valeurs, ni le checkpoint, ni les effets de bord. Le test doit comparer l'état complet après la première et la seconde exécution et démontrer qu'il reste identique.

4. Lire une proposition. Il faut connaître le temps CPU, les attentes PostgreSQL, le débit disque, les requêtes dominantes, la contention, l'occupation des autres files et la progression réelle. Si le traitement attend principalement `IO/DataFileRead`, davantage

de workers peuvent empirer la situation.
5. Écrire une mission. Une bonne mission précise le modèle concerné, la définition exacte de `stale`, les transitions autorisées, l'unicité et le rejeu. Elle demande une migration réversible, des tests de modèle et de job, puis un diff limité. Elle interdit de modifier les checkpoints stricts, les audits et les consommateurs sans proposition séparée.

22. Synthèse

Le développement assisté par IA ne supprime pas le métier de développeur. Il le déplace vers les responsabilités qui ont toujours déterminé la qualité réelle d'un logiciel :

- comprendre le problème ;
- définir la vérité ;
- séparer les responsabilités ;
- écrire des contrats ;
- limiter les changements ;
- contrôler les effets de bord ;
- exiger des preuves ;
- mesurer les performances ;
- conserver un historique explicable ;
- assumer les décisions.

La phrase à retenir

L'IA peut écrire le code. Le développeur-architecte décide ce que le système doit garantir, vérifie que ces garanties existent réellement et reste responsable de ce qui est publié.

23. Prochaine étape

Le chapitre suivant portera sur Ruby. Nous étudierons les objets, les classes, les méthodes, les modules, les collections, les blocs et les exceptions afin de comprendre la matière première élégante sur laquelle Rails est construit.

Pour poursuivre le voyage

Ces trois chapitres donnent un aperçu de la méthode du livre. L'édition complète contient **29 chapitres**, répartis en trois parties, pour un parcours de **404 pages** dans sa version PDF actuelle.

Vous y découvrirez notamment :

- le modèle UTXO, Bitcoin Core, le minage, les clés, les portefeuilles et les scripts Bitcoin ;
- les modules Layer1, Cluster, ActorProfile, ActorBehavior et ActorLabels ;
- Ruby, Rails, PostgreSQL, Active Record, les migrations, Sidekiq, Redis et Hotwire ;
- les tests, Git, l'observabilité, la sécurité et le refactoring ;
- une méthode concrète pour collaborer avec une IA sans lui abandonner les décisions importantes.

> **Le livre complet**
>
> Retrouvez Construire Tansa avec l'IA — Tome 1 : Conception et développement sur sa page Leanpub. L'achat donne accès au PDF et à l'EPUB complets ainsi qu'aux futures mises à jour de cette édition.

Merci pour votre lecture.

Victor Perez