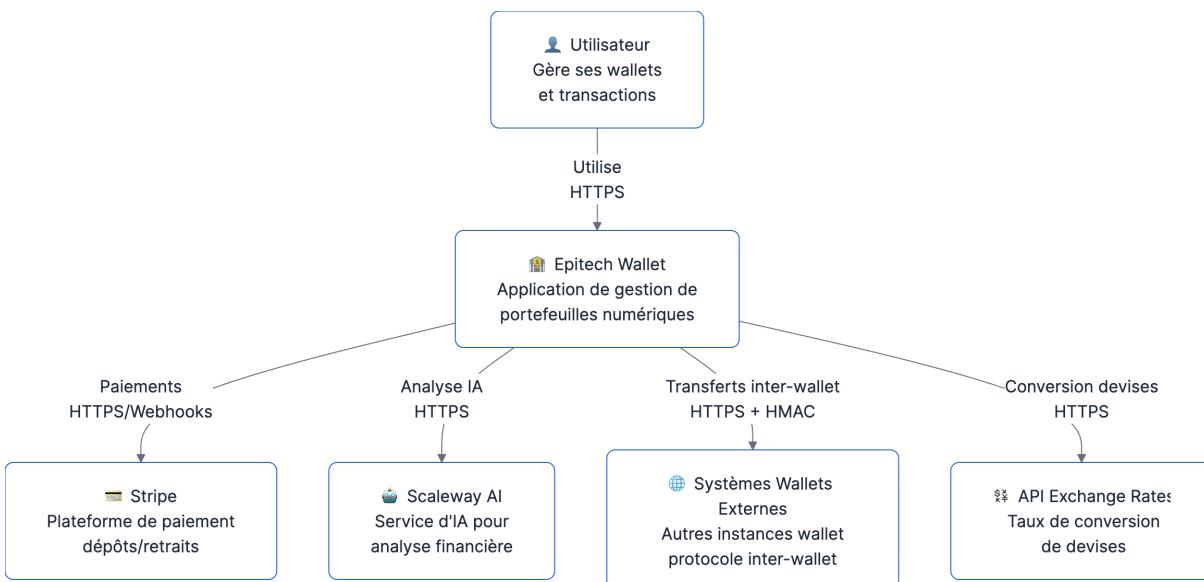


# Architecture technique

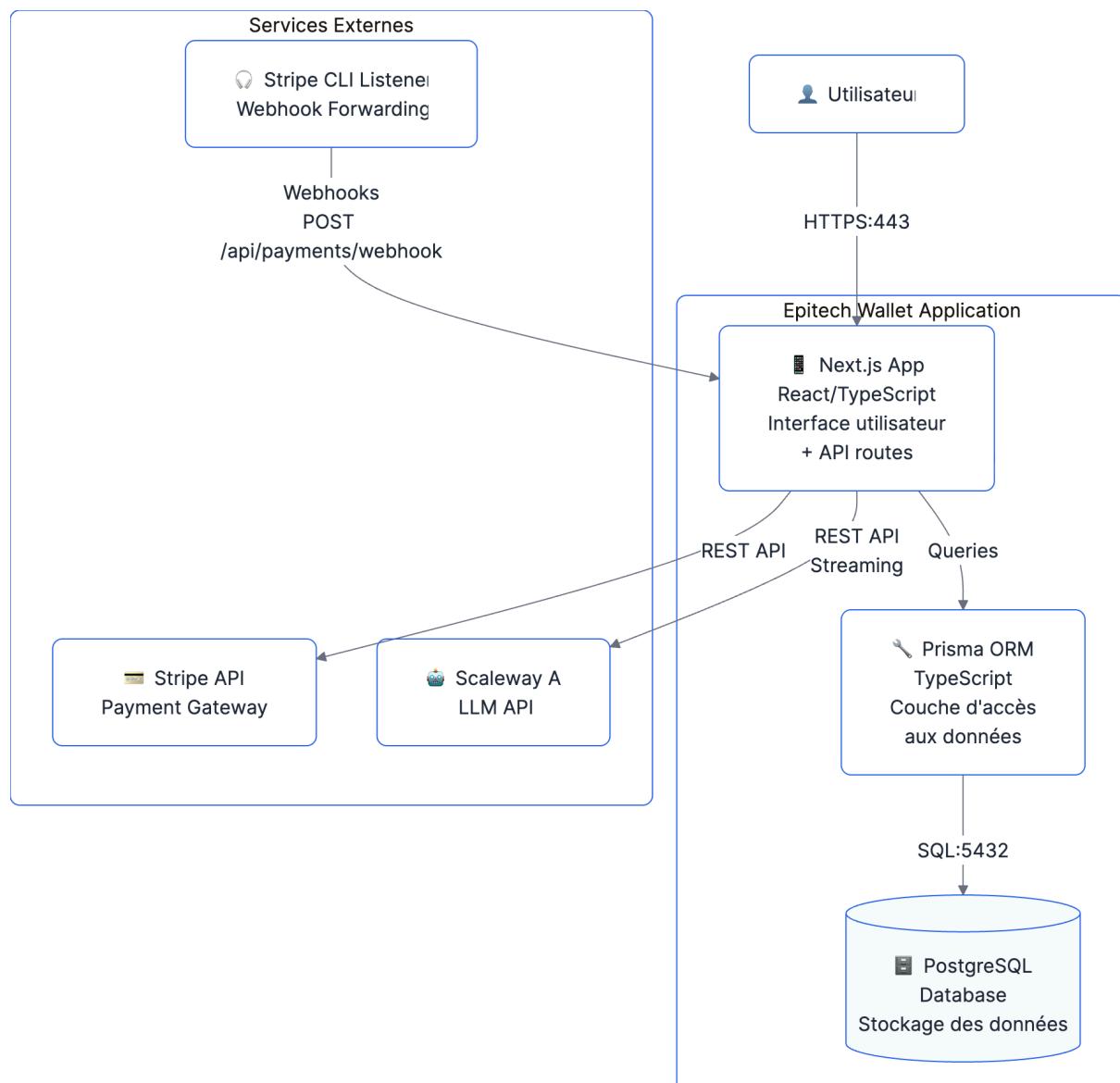
## Schéma 1 — Contexte système (Epitech Wallet)



Ce schéma fournit la vue "macro" du système. Côté utilisateur, le navigateur charge l'UI (Next.js/React) qui appelle les endpoints REST (API Routes) pour s'authentifier, lister les wallets, initier des transactions, lancer des dépôts Stripe ou des retraits.

Côté serveur, les routes API orchestrent la logique métier (calcul de la marge plateforme, contrôle de fraude, statuts de transaction), puis persévèrent l'état dans PostgreSQL via Prisma. Le schéma souligne aussi les intégrations : Stripe (paiement + webhook asynchrone), service de taux de change (conversion) et systèmes inter-wallet (échanges signés HMAC). Enfin, il met en évidence la traçabilité via les logs (TransactionLog / InterWalletLog) et l'importance de la cohérence des soldes (opérations atomiques en base).

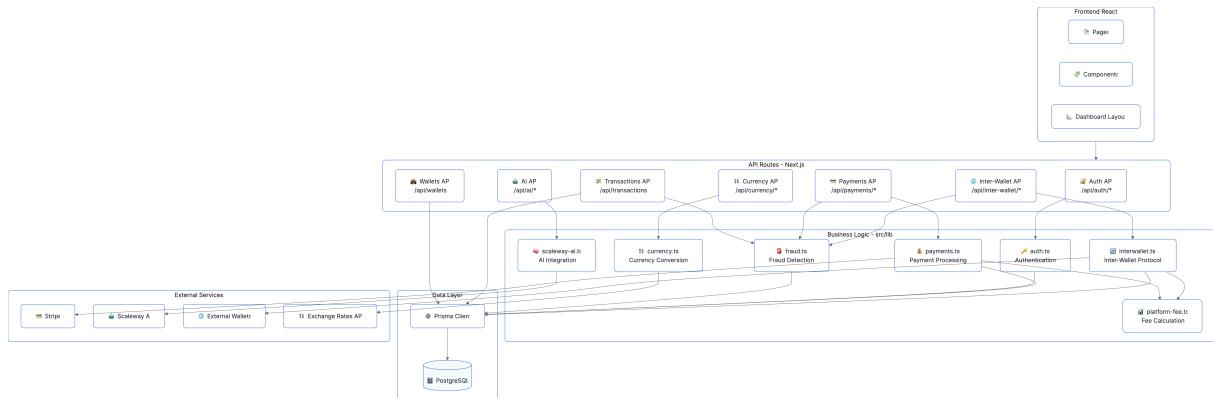
## Schéma 2 — Architecture containers



Ce schéma se concentre sur l'exécution "infra" : il met en scène l'application et ses dépendances techniques sous forme de conteneurs/services. On y voit typiquement l'application web (Next.js) exposée sur un port HTTP, la base PostgreSQL isolée derrière un réseau, et les volumes (ou mécanismes équivalents) qui garantissent la persistance des données.

L'intérêt est de clarifier les frontières : ce qui est interne au projet (app + DB) versus ce qui est externe (Stripe, API de change, autres wallets). Il sert aussi à expliquer le déploiement : variables d'environnement (DATABASE\_URL, secrets), connectivité réseau, et responsabilités de chaque brique (l'app orchestre, la DB stocke, les services externes fournissent paiement/FX/interop).

## Schéma 3 — Diagramme de composants (API & services)

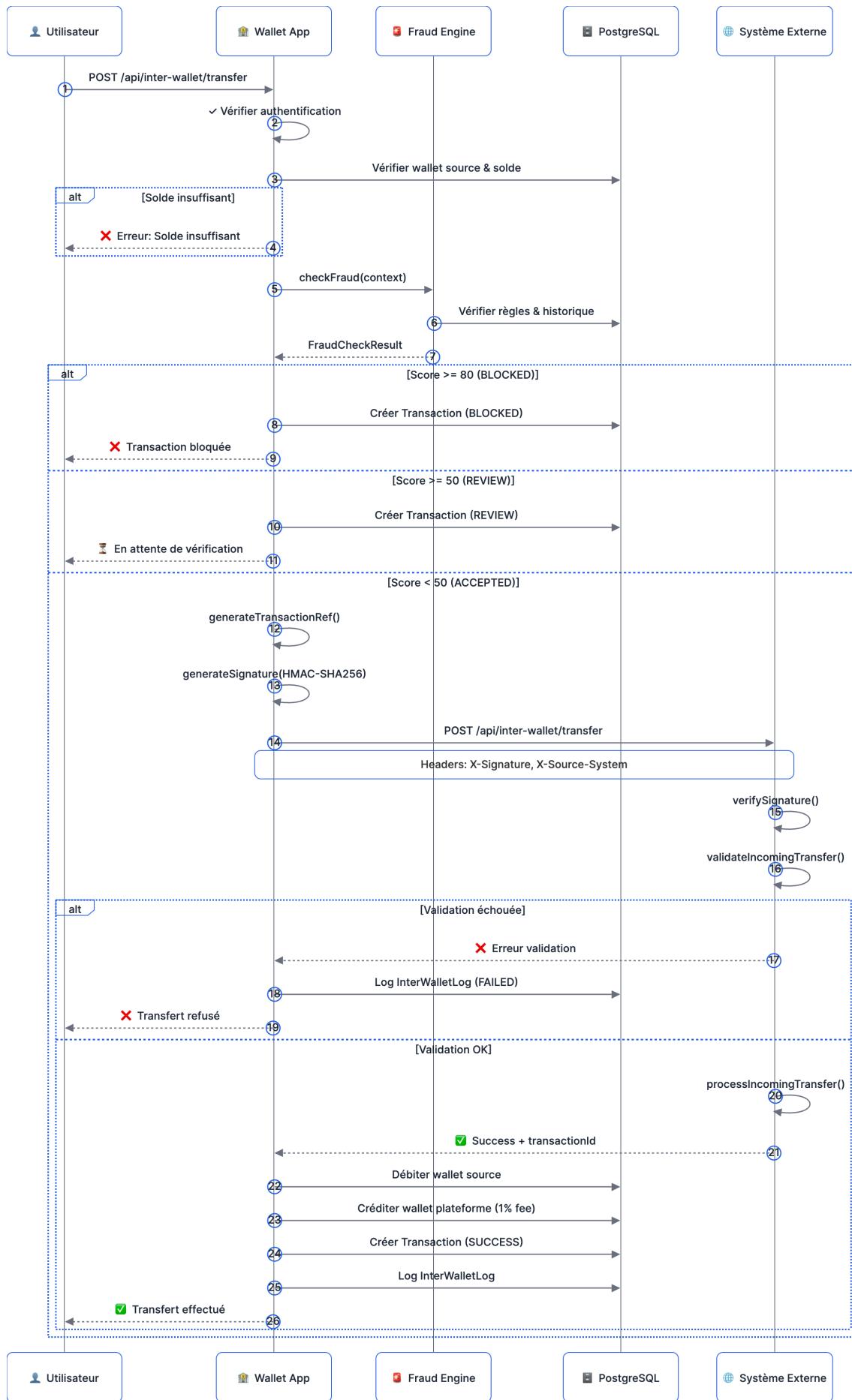


Ce schéma décrit la décomposition logique du back en composants. Il met en évidence les routes API (auth, wallets, transactions, paiements, inter-wallet) et les services transverses utilisés par ces routes : accès DB (Prisma), calcul de frais (platform-fee), conversion (currency), paiement (payments/stripe), fraude (fraud) et interop (interwallet).

En pratique, chaque endpoint valide les entrées (Zod côté routes), applique la logique métier et s'appuie sur des transactions Prisma (`prisma.$transaction`) pour garantir l'atomicité des opérations sensibles (débit/crédit + création de

transaction + logs). Le schéma sert donc à montrer où se trouve la logique, comment elle est réutilisée (via `src/lib/*`), et où sont les points d'intégration (Stripe, change, inter-wallet).

## Schéma 4 — Séquence : transfert inter-wallet sortant

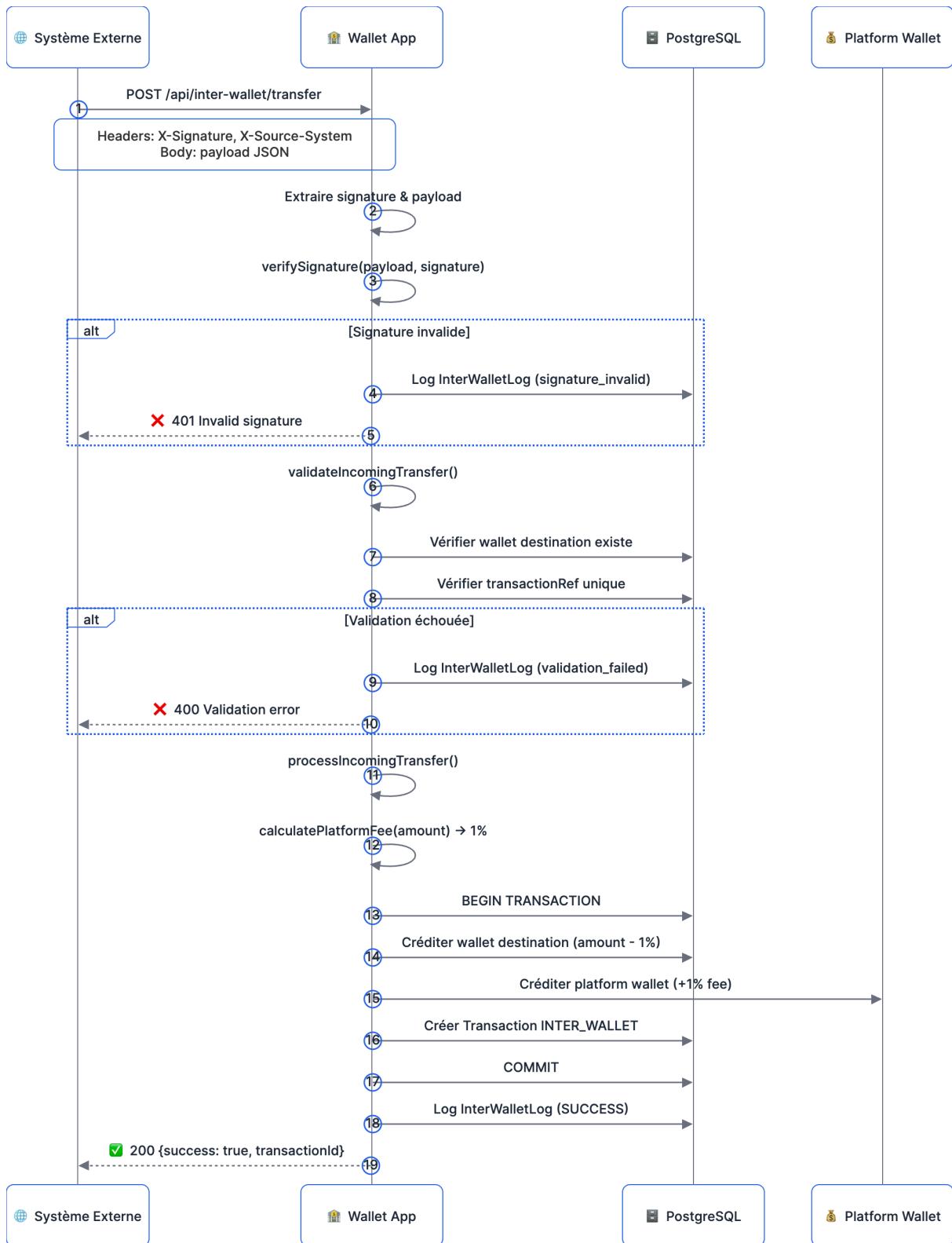


Ce schéma décrit la séquence d'un transfert inter-wallet sortant déclenché depuis notre application. Le flux démarre côté API par la validation de l'utilisateur (session), la vérification de la propriété du wallet source, puis l'exécution du contrôle anti-fraude.

Ensuite, le système calcule la marge plateforme (1%), débite le wallet source (montant + marge) et crédite le wallet plateforme dans une opération atomique, puis crée une transaction en base (type INTER\_WALLET) avec un statut initial PENDING. Une requête est alors envoyée vers le système distant (`/api/inter-wallet/transfer`) avec un payload horodaté et une signature HMAC fournie via `X-Signature`.

Si le système distant accepte la demande, la transaction passe à PROCESSING et une référence inter-système est stockée pour le suivi. En cas d'échec (rejet, timeout, erreur réseau), le système applique une compensation (remboursement du wallet source et annulation de la marge) et marque la transaction FAILED, en gardant l'historique des étapes pour l'audit et le débogage.

## Schéma 5 — Séquence : transfert inter-wallet entrant

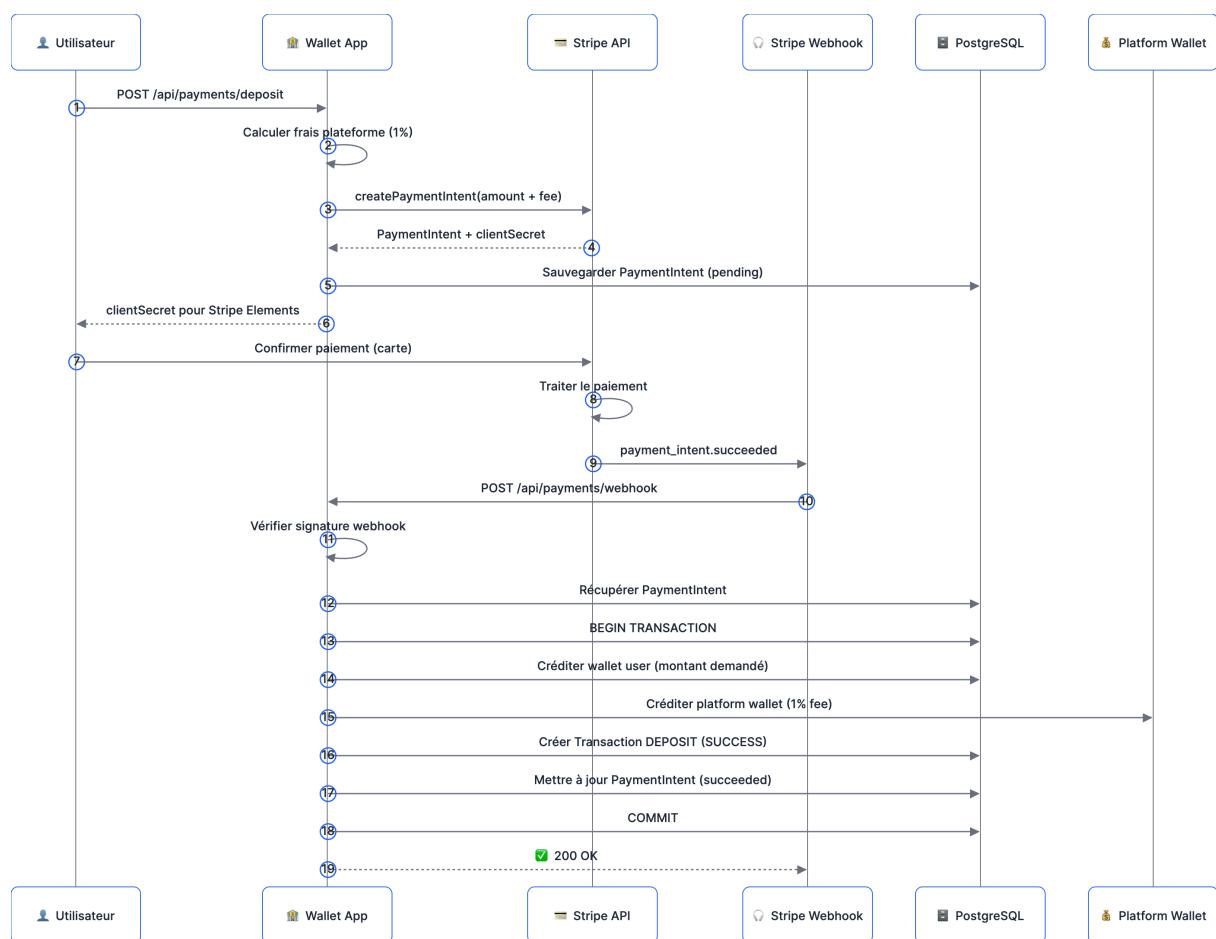


Ce schéma représente le traitement d'un transfert inter-wallet entrant (initié par un autre groupe). Le système reçoit un POST sur `/api/inter-wallet/transfer` avec un payload (référence, sourceSystemUrl, wallets, montant, timestamp) et une signature HMAC.

La première étape est la sécurité et l'idempotence: vérification de la signature (`x-Signature`), contrôle que le wallet destinataire existe et est actif, puis vérification que la référence inter-système n'a pas déjà été traitée (unicité) afin d'éviter les doubles crédits.

Une fois validé, le système crée une transaction en base (type INTER\_WALLET, statut SUCCESS) et met à jour les soldes de manière atomique: crédit du wallet destinataire du montant net (après marge) et crédit du wallet plateforme du montant de la marge. Le schéma se conclut par l'envoi d'un accusé de réception au système source, ainsi que l'écriture de logs d'échange inter-systèmes (payload, signature, réponse).

## Schéma 6 — Séquence : dépôt Stripe

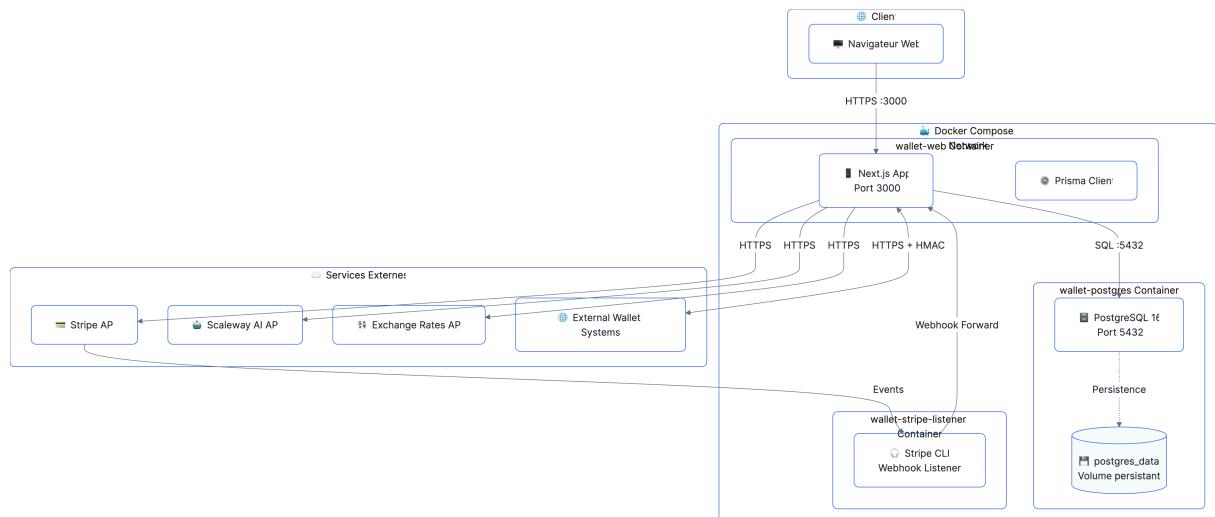


Ce schéma détaille le flux de dépôt via Stripe, qui est asynchrone par nature. Côté utilisateur, l'application crée une Checkout Session avec un total à payer qui inclut le montant à créditer, les frais Stripe et la marge plateforme. L'utilisateur est redirigé vers Stripe pour finaliser le paiement.

Côté serveur, la confirmation arrive via webhook: l'endpoint `/api/payments/webhook` vérifie d'abord la signature Stripe (sécurité critique), puis récupère le PaymentIntent Stripe et déclenche le traitement métier idempotent. Le système crédite le wallet utilisateur du montant attendu (le "montant à créditer"), crédite le wallet plateforme de la marge (avec conversion éventuelle vers EUR), et crée la transaction DEPOSIT ainsi que ses logs (PAYMENT\_RECEIVED, WALLET\_CREDIT, PLATFORM\_FEE, etc.).

Ce schéma met donc l'accent sur la séparation "paiement Stripe" vs "écriture comptable interne" et sur la nécessité d'idempotence pour éviter les doubles crédits en cas de redélivrance de webhooks.

## Schéma 7 — Infrastructure Docker Compose

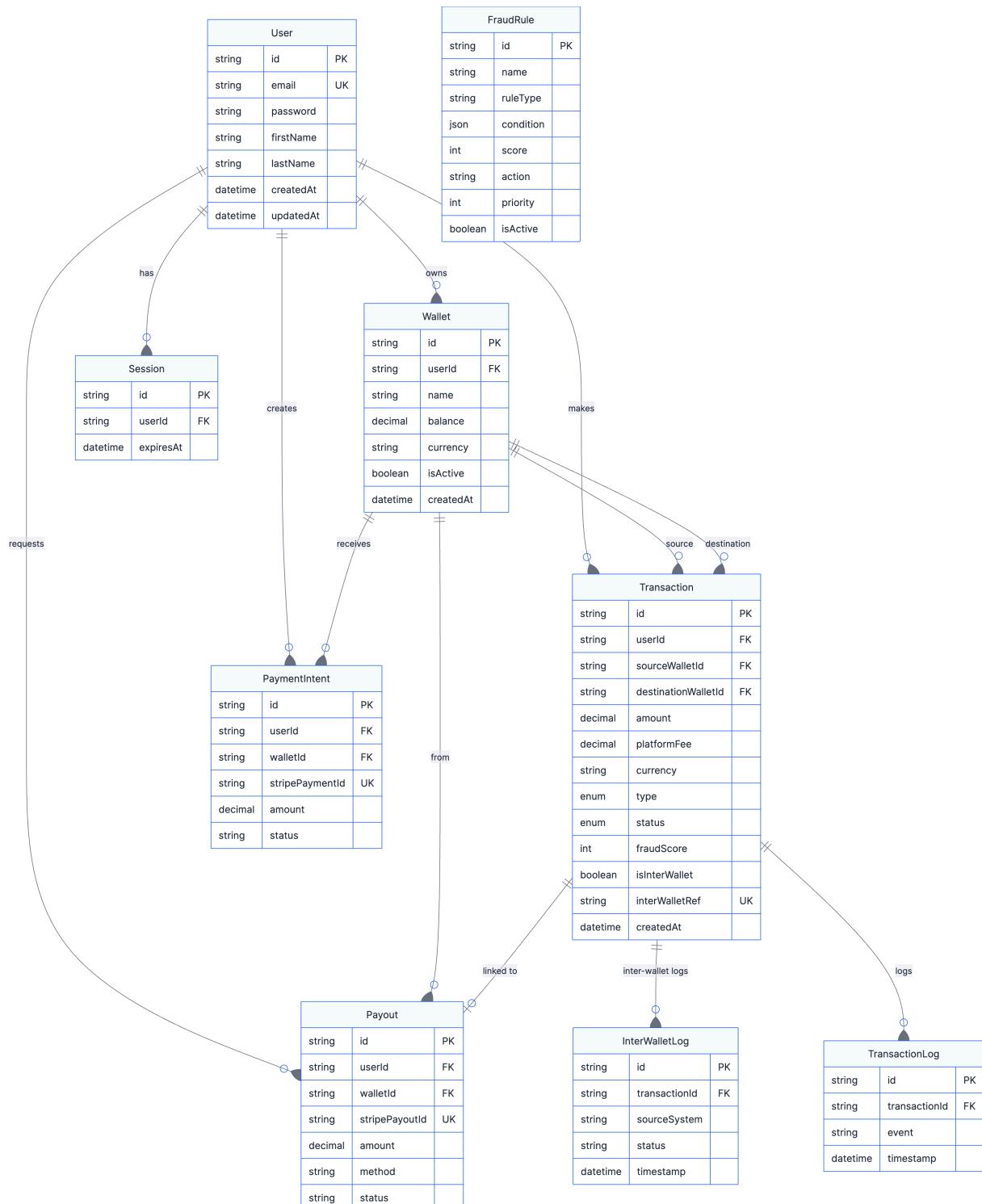


Ce schéma est la version opérationnelle de l'architecture de dev: il correspond au fichier `docker-compose.yml` et décrit précisément les services lancés en local. Le

service PostgreSQL y expose un port, reçoit ses identifiants, et stocke ses données sur un volume pour survivre aux redémarrages.

L'application, elle, se connecte à la DB via `DATABASE_URL` (Prisma) et dépend de ce service pour toutes les fonctionnalités cœur (wallets, transactions, logs, règles de fraude). Le schéma rappelle aussi les points à configurer pour les intégrations (secrets Stripe, secrets HMAC inter-wallet) et permet de diagnostiquer rapidement une panne (port indisponible, volume manquant, variables non définies).

## Schéma 8 — Modèle de données (ERD)

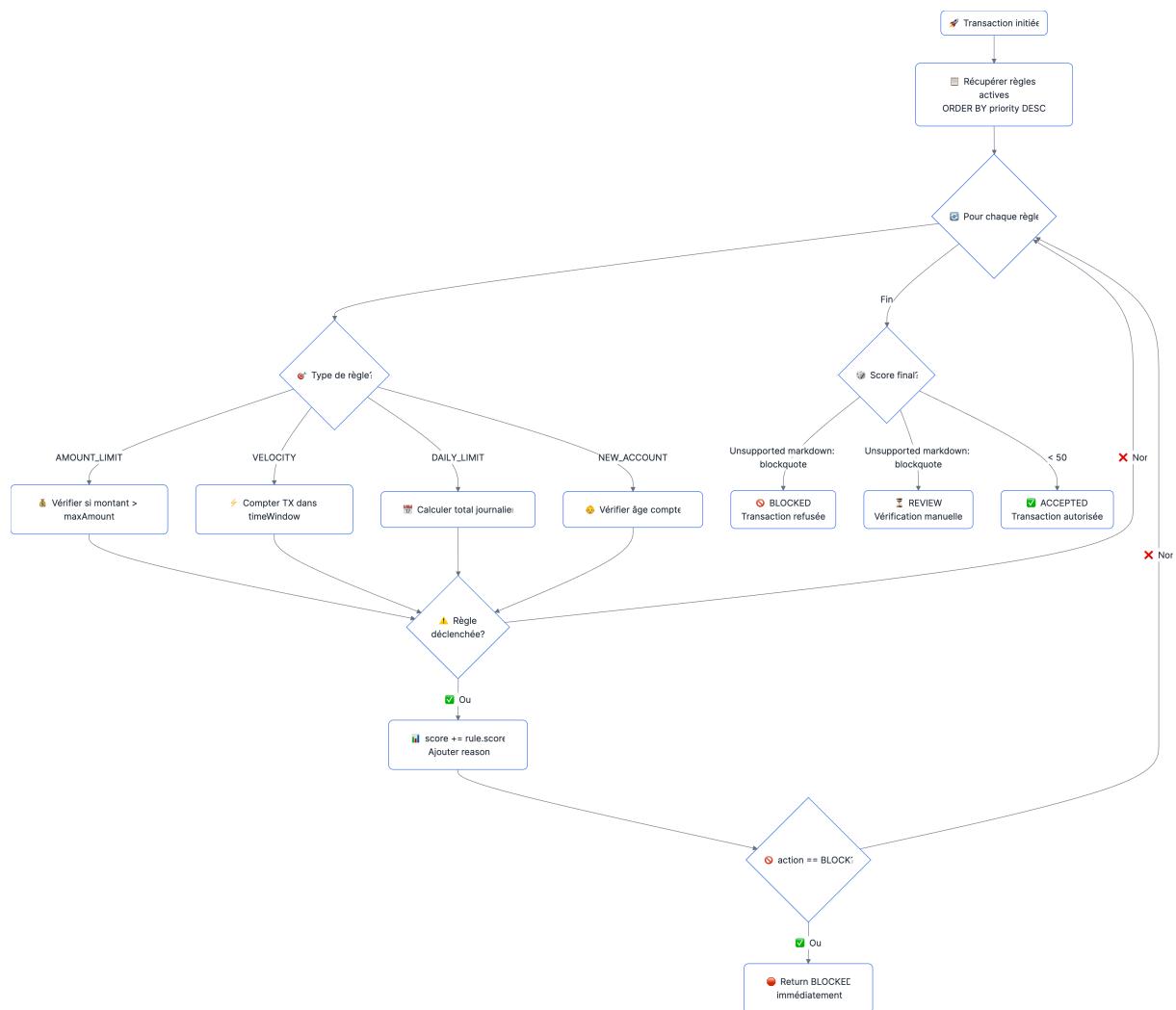


Ce schéma (ERD) décrit le modèle de données et les liens qui supportent la logique comptable et la traçabilité. Un **User** possède plusieurs **Wallet** (soldes, devise), et les mouvements sont enregistrés sous forme de **Transaction** (type, statut, montants, frais, champs fraude, métadonnées).

Les relations `sourceWalletId` / `destinationWalletId` permettent de représenter les transferts, tandis que `TransactionLog` capture les étapes techniques (validation, fraude, débit, crédit, frais) pour audit. Pour l'interop, `InterWalletLog` conserve les payloads signés, les signatures et les réponses externes. Le schéma inclut aussi `PaymentIntent` (dépôts Stripe) et `Payout` (retraits) pour relier les flux de paiement à des transactions internes, ainsi que `FraudRule` pour les règles de scoring.

Enfin, les index/constraintes (par ex. unicité de `interWalletRef`) servent à garantir l'idempotence et la cohérence des traitements (éviter les doubles crédits, retrouver rapidement l'historique d'un wallet ou d'une transaction).

## Schéma 9 — Flux de détection de fraude

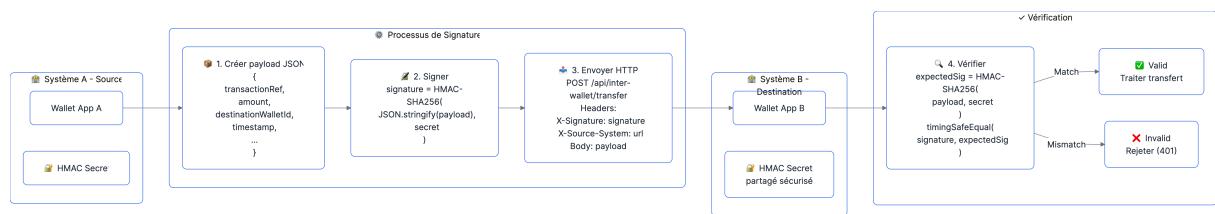


Ce schéma formalise la logique de décision anti-fraude. Lorsqu'une transaction est demandée, le backend construit un contexte (userId, montant, type, wallets, inter-wallet ou non) et évalue des règles actives (chargées depuis la base, triées par priorité). Chaque règle peut ajouter des points au score et fournir une raison; certaines règles peuvent bloquer immédiatement.

Si aucune règle personnalisée n'existe, un set de règles "built-in" s'applique (montants élevés, vélocité, limite journalière, compte récent). Le score final est converti en décision avec des seuils simples: ACCEPTED (<50), REVIEW (50–79), BLOCKED ( $\geq 80$ ). La décision impacte directement le statut de la transaction: BLOCKED empêche l'exécution, REVIEW place la transaction dans un état intermédiaire (à traiter), SUCCESS exécute le débit/crédit.

Le schéma met aussi en avant la transparence: score et raisons sont stockés sur la transaction ( `fraudScore` , `fraudReason` ) et visibles dans l'historique, ce qui aide à expliquer les blocages et à tester/démontrer le moteur.

## Schéma 10 — Protocole inter-wallet : sécurité HMAC



Ce schéma explique la couche de sécurité du protocole inter-wallet. Le principe est celui d'un secret partagé ( `INTERWALLET_HMAC_SECRET` ) entre systèmes: l'émetteur calcule un HMAC-SHA256 sur le JSON du payload (transactionRef, sourceSystemUrl, wallets, montant, timestamp...) et l'envoie dans le header `Signature`.

Le receveur recalcule la signature à partir du payload reçu et compare en timing-safe (pour limiter les attaques par timing). Si la signature ne correspond

pas, la requête est rejetée. Si elle correspond, le receveur peut traiter la demande et journaliser l'échange (payload, signature, statut HTTP, réponse) dans `InterWalletLog`.

Le schéma montre aussi l'enchaînement fonctionnel des endpoints: `transfer` pour initier/réceptionner, `validate` pour confirmer/rejeter, et `status` pour consulter l'état d'une transaction. Le champ `timestamp` fournit un support à la protection anti-rejeu, même si la fenêtre de validité (et un nonce) reste à définir/implémenter si on veut renforcer la sécurité.