TAURACO · FINANCEMENT INSTITUTIONNEL & ERP OHADA

Documentation technique Tauraco

Documentation technique

Édition du 25 septembre 2026

tauraco.io · app.tauraco.io

Sommaire

Tauraco est la plateforme de gestion et de financement des entreprises de l'espace OHADA : un ERP complet (facturation, comptabilité SYSCOHADA, trésorerie, stocks, immobilisations), la facturation électronique, et un financement institutionnel intégré — vos factures comme vos marchandises, instruites par des banques partenaires qui fixent leurs propres conditions, avec sûretés et tierce détention de niveau bancaire. Ce document sert deux usages : diagnostiquer un incident (support) et faire évoluer le code sans régresser un invariant métier (évolutions). Chaque affirmation ci-dessous a été vérifiée dans le code au moment de la rédaction — les chiffres (migrations, fichiers de routes, types d'alerte, tables) sont recomptés, pas recopiés depuis une documentation archivée. Quand une divergence existait entre une page archivée et le code, c'est le code qui fait foi et l'écart est signalé explicitement.

> Aucun secret n'apparaît dans ce document. Les variables d'environnement sont citées par leur **nom** seulement (ex. `DEV_AUTH_SECRET`) — jamais leur valeur.

01 Vue d'ensemble

1.1 Les quatre surfaces

SURFACE	STACK	RÔLE	ORIGINE RÉELLE SUR CETTE MACHINE
`tauraco.ai`	Next.js 14 (`frontend-ai/`)	Vitrine marketing — financement institutionnel	conteneur `frontend_ai` :3000, derrière nginx :8088
`tauraco.io`	HTML/JS statique (`frontend/`)	Vitrine ERP + pages historiques (login, dashboard, marketplace legacy) + alias `/app`	fichiers statiques servis par nginx, + proxy `/auth/*`, `/v1..v3/*` vers le backend
`app.tauraco.io`	Vue 3 + Vite + PrimeVue + Pinia (`frontend-app/`)	SPA cible en production — surface canonique	conteneur `tauraco-app-web` (nginx), projet compose séparé `docker-compose.app.yml`, exposé sur `127.0.0.1:18190`
`api.tauraco.io`	FastAPI + uvicorn (`backend/`)	API REST, ~74 modules de routes	conteneur `backend` :8000, derrière nginx :8088

1.2 Stack technique

- **Backend** : FastAPI, Python 3.12, SQLAlchemy, PostgreSQL 16, Redis 7 (Streams), Cloudflare R2 (S3-compatible) en prod / MinIO en dev, ClamAV pour l'antivirus.
- **SPA app** : Vue 3 (Composition API) + Vite + TypeScript + PrimeVue + Pinia + Vue Router + vue-i18n + Vitest + Playwright.
- **Mobile** : deux applications Expo/React Native (`tauraco-erp`, `tauraco-financement`) — voir §6.

1.3 Topologie de déploiement réelle

Deux chemins d'entrée coexistent sur cet hôte, et ne pas les confondre évite des heures de diagnostic :

```
Internet
  │
  ▼
Cloudflare (TLS/CDN)
  │
  ├── chemin LiteSpeed (hébergement partagé, api.tauraco.io / tauraco.io / tauraco.ai)
  │     └ LiteSpeed ──► nginx (127.0.0.1:8088, conteneur "nginx", docker-compose.yml)
  │           ├ tauraco.ai      → frontend_ai:3000 (Next.js)
  │           ├ api.tauraco.io  → backend:8000 (FastAPI)
  │           └ tauraco.io      → statique + proxy backend (/auth/*, /v1../v3/*)
  │
  └── chemin Cloudflare Tunnel (cloudflared.service, systemd, /etc/cloudflared/config.yml)
        ├ app.tauraco.io            → 127.0.0.1:18190 → conteneur tauraco-app-web
        │    (projet compose séparé : docker-compose.app.yml ; docroot = symlink
        │     .app-release → .app-releases/<horodatage>-<sha>/)
        ├ collateral-demo.tauraco.io → 127.0.0.1:18180 → tauraco-collateral-demo-web-1
        │    (DEMO_FICTIF, backend + Postgres isolés — jamais servi à un vrai client)
        └ agent.maliactu.net        → (service séparé)
```

Le backend ne recharge jamais à chaud. Le code est monté en bind-mount (`./backend:/app`) mais uvicorn tourne **sans** `--reload` . Toute modification de code backend exige :

```
docker compose restart backend
```

Un `docker compose up -d` après changement de `.env` ne suffit pas non plus à propager une variable dans un conteneur déjà démarré — il faut recréer le service concerné, pas seulement le redémarrer, si la variable est lue à l'import (voir §5, « env modifié »).

nginx fige l'IP du backend. `nginx/default.conf` déclare `proxy_pass http://backend:8000` sans directive `resolver` : nginx résout le nom de service Docker une fois (au chargement de la configuration) et garde l'IP en mémoire pour toute la durée du process. Si le conteneur `backend` est **recréé** (nouvelle IP dans le réseau `tauraco_net`) — par opposition à un simple `restart` , qui conserve l'IP —, nginx continue de parler à l'ancienne IP jusqu'à son propre redémarrage :

```
docker compose up -d --force-recreate backend
docker compose restart nginx   # obligatoire après une recréation, pas après un resta
```

1.4 Déployer la SPA `app.tauraco.io`

Procédure et scripts documentés dans `docs/runbooks/deploy-frontend-app.md` :

```
./scripts/deploy_app_canonical.sh        # build + promotion atomique (surface canonique a
./scripts/deploy_app_alias.sh            # même commit, base "/app/" — alias tauraco.io/ap
./scripts/check_frontend_drift.sh --expect-head   # gate anti-drift
```

`deploy_app_canonical.sh` construit avec `vite build --mode canonical` , applique quatre garde-fous (base `/` , présence de `api.tauraco.io` dans le bundle, aucune URL `localhost` / `127.0.0.1` / `backend:` , `build_id` extractible), copie le résultat dans `.app-releases/<horodatage>-<sha>/` , bascule le symlink `.app-release` , puis **recrée le conteneur** — étape non optionnelle : Docker résout le symlink au moment du montage, donc basculer le lien sans recréer le conteneur ne change rien à ce qui est servi.

`check_frontend_drift.sh` compare le `/build-info.json` (public, sans secret) de la surface canonique et de l'alias, et avec `--expect-head` , au SHA git courant. Codes de sortie : `0` conforme, `1` drift, `2` la surface canonique ne publie pas de `build-info.json` . Les deux surfaces doivent être construites **depuis le même commit** —

c'est exactement l'incident du 2026-08-22→09-01 qui a motivé ce gate (l'ingress `app.tauraco.io` pointait par erreur sur l'origine du collateral demo, jamais branchée au pipeline de release).

L'alias `tauraco.io/app` reste nécessaire tant que `PUBLIC_APP_URL` (`backend/app/core/config.py`) compose des URL de retour Stripe sur `https://tauraco.io/app/...` (`api/routes/tenant_subscription.py`, `api/routes/payment_admin.py`) — le retirer avant de basculer cette base casserait un flux de paiement vivant. La procédure de retrait est documentée dans le runbook, dans l'ordre.

1.5 Domaines métier (préfixes de route)

PRÉFIXE	SIGNIFICATION
(sans préfixe)	Routes historiques (auth, `/invoices`, `/companies`)
`/v1/…`	ERP (billing, AP, AR, trésorerie, immobilisations, paiements)
`/v2/…`	Pipeline facture, admin, ops, IA (`/v2/ai/*`)
`/v3/…`	Portails, FX, training ML, notifications, drift, financement institutionnel (`/v1/financing/*` et `/v3/*` selon le module)

02 Backend

2.1 Organisation

Le backend regroupe **74 fichiers de routes** dans `app/api/routes/` (un `APIRouter` par fichier) plus `app/domains/signatures/routes.py`, et **74 services** (`app/services/*_service.py`). Deux axes d'organisation se superposent :

- **Par phase produit** — les fichiers portent parfois un suffixe (`invoices_v2.py` vs `invoices_v2_p2.py`) qui distingue des révisions successives de la même surface plutôt que des domaines différents.
- **Par domaine** — `erp_*` (12 fichiers : accounting, assets, billing, comments, dashboard, expert_comptable, french, fx, import, ingestion, inventory, lettrage, migration, posting, recurring, settings, suppliers, treasury), `portal_*` (admin, seller), `payment_*` (admin, gateway, links, payments), `financing_*` (bank, borrowing_base, company, facilities, financing), `collateral*`, `invoices_v2*`.

`app/domains/` regroupe le code métier structuré par capacité plutôt que par type de fichier : `collateral/` (registre, RLS, parcours, alertes, exploitation, référentiel), `contracts/`, `financing/` (32 fichiers dont `marketplace.py` — calcul de position concurrentielle, **n'enregistre aucune route**), `investors/` (contient le gel legacy — voir §2.2), `legal_review/`, `mandate/`, `ohada_reference/`, `signatures/`.

Convention pour une nouvelle fonctionnalité (checklist complète en §7) : modèle ORM (`domain/models.py`) → enum (`domain/enums.py`) → migration Alembic → schéma Pydantic (`schemas/`) → service (`services/<nom>_service.py`, prend une `Session`) → route (`Depends(require_roles(...))`) → inclusion du router dans `main.py` → tests. Les routes doivent rester minces — si de la logique s'accumule dans une route, vérifier qu'un service existant ne devrait pas la porter.

2.2 Le pipeline facture

```
UPLOADED → FILE_PREPROCESSED → OCR_DONE → FIELDS_EXTRACTED → VALIDATED → SCORED
```

Chaque transition est pilotée par un worker consommant Redis Streams (`xreadgroup`). `SCORED` est l'**état terminal actuel** du pipeline.

ÉTAPE	ÉMIS PAR	CONSOMMÉ PAR	ACTION
`FILE_UPLOADED`	API `/finalize-upload`	worker-antivirus	fichier enregistré
`FILE_PREPROCESSED`	worker-antivirus	worker-pipeline	scan ClamAV OK
`OCR_DONE`	worker-pipeline	worker-pipeline	OCR (DocTR/Tesseract/mock)
`FIELDS_EXTRACTED`	worker-pipeline	worker-pipeline	ExtractionService
`INVOICE_VALIDATED`	worker-pipeline	worker-pipeline	FraudService + ValidationService
(scoring)	worker-pipeline	—	ScoringService + FeatureBuilderService → `Invoice.status = SCORED`

Consommation Redis : `redis.xreadgroup(group, consumer, {stream: ">"}, count=count, block=block_ms)` dans `app/infra/queue.py` (classe `EventBus`). XACK explicite seulement au succès du handler ; après `DLQ_MAX_RETRIES=3` échecs, le message part dans `tauraco:stream:dlq` puis est ACK.

> **Ce qui a changé depuis le Lot 1 (2026-09)** — l'étape suivante publiait autrefois la facture sur un marketplace investisseur privé (`MarketplaceService` , `INVOICE_LISTED` / `OFFER_CREATED` /allocation/ `INVOICE_FUNDED`). Cette chaîne a été déposée (`docs/legal/2026-09-14-fermeture-consultation-investisseur.md`) : le service est supprimé, les tables (`investors` , `investor_wallets` , `market_listings` , `allocations` , `offers` legacy, `auctions` , `bids`) restent intactes en base (aucun solde touché) mais gelées. Les valeurs d'événement correspondantes (`SCORE_READY` , `INVOICE_LISTED` , `OFFER_CREATED` , `ALLOCATION_RESERVED` , `ALLOCATION_CONFIRMED` , `INVOICE_FUNDED`) restent dans l'enum `EventType` et sont acquittées comme no-op terminal par `pipeline_worker` — uniquement pour ne jamais faire échouer l'acquittement d'un message résiduel déjà présent dans le stream au moment de la dépose. Les factures scorées alimentent désormais le circuit de financement institutionnel (banques partenaires) via `FinancingService` .

Diagnostiquer une facture bloquée

1. `GET /v2/ops/dlq` (rôle ADMIN) — voir les messages en dead-letter.
2. Vérifier les logs du worker concerné : `docker logs tauraco-worker-pipeline-1 --tail 100 -f` (ou `worker-antivirus` si le blocage est en `UPLOADED` / `PREPROCESSING`).
3. `redis-cli XINFO GROUPS tauraco:stream:invoices` — le groupe consumer a-t-il perdu sa place ?
4. Corriger la cause racine, puis `POST /v2/ops/dlq/{event_id}/retry` .
5. En dernier recours, forcer le statut : `POST /v2/ops/invoices/{id}/force-status` .

Une facture bloquée signifie presque toujours qu'un worker a planté ou que son consumer group a perdu sa place — vérifier les logs du worker avant de toucher directement à la base.

2.3 Migrations Alembic

86 migrations numérotées (`001_initial_schema.py` → `086_conditions_structurees.py`), tête actuelle `086` . La constante `TETE_ATTENDUE` (déclarée dans trois fichiers de test — `test_institutional_baseline.py` ,

`test_migration_harness.py`, `test_production_baseline_v2.py`) fait échouer la suite si une migration a été ajoutée sans mettre à jour la référence attendue.

Toutes les migrations sont **idempotentes** (gardes `IF NOT EXISTS`/inspection avant `create_table`/`add_column`) — `create_all_tables()` tourne au démarrage du backend, donc rejouer `alembic upgrade head` sur une base déjà à jour ne doit jamais échouer.

JALON	CONTENU
030	Rôle PG `tauraco_requeteur` — SELECT seul + `default_transaction_read_only=on` + `statement_timeout=5s`
043–050	Collateral phase 1-4 : registre, RLS + vues, parcours, alertes, projections
051–056	Financement institutionnel : demande, offres, éligibilité
053	Rôle applicatif `tauraco_financing_app`
067–068	Commitment/CreditFacility puis Drawdown/Disbursement/Repayment — Tauraco enregistre les faits bancaires, il ne paie pas
073	Axe réel/démo sur Tenant et espaces financeurs
074–077	Registre de servicing (ajout seul), vocabulaire d'alerte réparé, seuils configurables
078–082	Sous-ledger de stock (9 tables), assiette versionnée, custodian acteur, sorties de stock
084	KYB institution (`CollateralInstitutionKybCase`, distinct du KYB vendeur)
085	`FinancingRequestAnalysis` versionnée + abonnement institution
086	**Tête actuelle.** Conditions d'offre structurées (`rate_bps`, `rate_type`, `advance_rate_bps`, `fees_bps`) sur `FinancingOffer`

> **Ne jamais modifier une migration déjà appliquée en production.** Toujours créer une nouvelle migration numérotée en séquence. L'historique du projet porte la trace de plusieurs correctifs prod livrés ainsi (070, 071, 083 — droits de rôle oubliés dans une migration précédente, jamais réécrits sur place).

Exécution :

```
# avant de redémarrer le backend (zero-downtime)
docker exec tauraco-backend-1 alembic upgrade head
docker compose restart backend
```

Pattern idempotent recommandé :

```
bind = op.get_bind()
insp = inspect(bind)
if "my_table" not in insp.get_table_names():
    op.create_table("my_table", ...)
cols = [c["name"] for c in insp.get_columns("existing_table")]
if "new_column" not in cols:
    op.add_column("existing_table", sa.Column("new_column", sa.String))
```

`prepare_pre_migration_schema.py` (utilisé par `test_contrat_privileges_postgres.py` et par le job CI « Migrate a fresh PostgreSQL 16 to head ») pré-monte un schéma jusqu'à une révision donnée sans passer par les migrations elles-mêmes, pour contourner un défaut antérieur à Collateral : `alembic upgrade head` depuis une base réellement vide échoue dès la migration `002` (`invoices.id` créé en `String(50)` par la `001`, alors que la `002` et l'ORM le veulent en `BigInteger`). La CI l'utilise pour prouver qu'un segment de

migrations (ex. `041→044`) s'applique proprement sur un PostgreSQL 16 neuf, sans dépendre de ce défaut historique.

03 Sécurité et isolation

> Cette section corrige plusieurs pages archivées (`docs/backend/04-auth-rbac.md`, `docs/backend/09-security.md`, et l'état « à traiter avant prod » encore inscrit dans `CLAUDE.md`) : **la migration RS256 est faite dans le code** (`backend/app/core/security.py`), pas seulement préparée. Vérifier `app/core/security.py` en cas de doute plutôt que ces pages.

3.1 Authentification — RS256

Les jetons d'accès sont signés en **RS256** (clé privée) et vérifiés avec la clé publique — plus de secret partagé pour la signature. Les clés vivent dans `secrets/jwt_private.pem` / `secrets/jwt_public.pem`, montées en lecture seule sur `/run/secrets` dans les conteneurs, et lues via `JWT_PRIVATE_KEY_PATH` / `JWT_PUBLIC_KEY_PATH` (ou leur contenu inline, `JWT_PRIVATE_KEY` / `JWT_PUBLIC_KEY`, pour les tests). `load_jwt_keys()` est appelée depuis le lifespan de `main.py` avant que le service accepte du trafic : si la paire est absente ou malformée, `JWTKeyError` est levée et le **boot s'arrête** — jamais de repli HS256 silencieux. Chaque jeton porte un `kid` (`JWT_KEY_ID`) vérifié au décodage.

Tolérance HS256 legacy, hors production uniquement. `decode_access_token()` (`app/core/security.py`) accepte un jeton HS256 signé avec `DEV_AUTH_SECRET` *seulement si* `settings.JWT_ACCEPT_LEGACY_HS256` est vrai **et** `ENV` n'est ni `prod` ni `production` :

```
legacy_allowed = settings.JWT_ACCEPT_LEGACY_HS256 and env not in ("prod", "production")
if header.get("alg") == "HS256" and legacy_allowed:
    return jwt.decode(token, legacy_secret, algorithms=["HS256"])
raise ValueError("JWT algorithm is not accepted")
```

Cette tolérance existe pour laisser cohabiter, le temps que la flotte converge après un déploiement RS256, des sessions déjà munies d'un jeton HS256 (durée de vie ≤ 15 min) avec les nouveaux jetons RS256 émis par `/auth/refresh` et `/auth/dev-login`. Elle doit être refermée (`JWT_ACCEPT_LEGACY_HS256=false`) dans les 24h suivant un déploiement — **ce n'est pas automatique**, un opérateur bascule le réglage.

En production, ce repli n'existe jamais, quelle que soit la valeur du réglage — vérifié à deux endroits indépendants :

- `decode_access_token()` ignore `JWT_ACCEPT_LEGACY_HS256` dès que `ENV` vaut `prod` / `production` ;
- `validate_production_security()` (appelée au boot par le lifespan, après `validate_production_security`) lève `RuntimeError` si `JWT_ACCEPT_LEGACY_HS256=true` est trouvé en production — ceinture-et-bretelles : un déploiement mal configuré échoue fort plutôt que de masquer un réglage mort.

`validate_production_security()` vérifie aussi, en production : `DEV_AUTH_ENABLED=false`, `TURNSTILE_QA_BYPASS_ENABLED=false`, et que `CORS_ALLOW_ORIGINS` correspond exactement à l'allow-list Tauraco (`https://tauraco.io`, `https://www.tauraco.io`, `https://app.tauraco.io`, `https://tauraco.ai`, `https://www.tauraco.ai`).

Allowlist des administrateurs fondateurs — `GOOGLE_ADMIN_EMAILS`

Sur `POST /auth/google`, un compte Google vérifié sans aucun rôle en base ne reçoit `SUPER_ADMIN` que si son e-mail figure dans `GOOGLE_ADMIN_EMAILS` (liste explicite, normalisée en minuscules). Tout autre compte sans rôle est refusé (`403 COMPTE_NON_PROVISIONNE`) — et si l'upsert avait activé ou créé la ligne `User` avant que la décision d'accès ne soit connue, l'activation est annulée (`user.is_active = False`) pour ne jamais laisser un orphelin actif.

3.2 JTI, dénylistage, rotation de refresh, lockout

MÉCANISME	DÉTAIL
JTI + dénylist	`POST /auth/logout` écrit `tauraco:token:denied:{jti}` dans Redis (TTL = expiration du jeton). Vérifié à chaque requête dans `get_current_user` (`core/deps.py`). **Fail-open** si Redis est indisponible — choix de disponibilité assumé.
Refresh — corps de requête	`POST /auth/refresh` lit `refresh_token` dans le **corps JSON** (`RefreshRequest.refresh_token`, `app/api/auth.py`) — **pas** en Bearer, contrairement à ce qu'indique `docs/backend/04-auth-rbac.md` (page archivée).
Rotation + révocation de famille	`RefreshSessionService` (`app/services/refresh_session_service.py`) stocke un hash sha256 du refresh token, jamais le token brut. Chaque `rotate()` marque la session `used_at` et chaîne `replaced_by_id` vers la nouvelle. Si un token déjà utilisé ou révoqué est rejoué, **toute la famille** (`family_id`) est révoquée (`revoke_family()`) et l'appel lève « Refresh token replay detected » — la détection de rejeu invalide toute la chaîne de rotation, pas seulement le jeton rejoué.
Lockout	5 tentatives de login échouées (`LOCKOUT_MAX_ATTEMPTS`) → compte bloqué 15 min (`LOCKOUT_WINDOW_SECONDS=900`), compteur Redis `tauraco:lockout:{username}`.
Mots de passe	`sha256_crypt` via passlib (pas bcrypt — incompatibilité passlib/Python 3.12).

3.3 Multi-tenant

`tenant_id` (string) est présent en base et dans le JWT. Chaque endpoint doit filtrer sur `user.tenant_id` — aucune requête ne doit pouvoir traverser les tenants. `require_roles(allowed_roles)` (`core/deps.py`) résout l'utilisateur, vérifie `has_any_role` (toujours vrai pour `SUPER_ADMIN`), et renvoie 403 sinon. **19 rôles** au total : 11 rôles plateforme (`SUPER_ADMIN`, `ADMIN`, `OPS`, `RISK`, `FINANCE`, `INVESTOR_MANAGER` — gelé, `AUDITOR`, `INVESTOR` — gelé, `SELLER`, `COMPLIANCE`, `FINANCEUR`) + 8 rôles ERP par tenant (`ERP_OWNER` → `ERP_VIEWER`). `FINANCEUR` n'ouvre par lui-même aucune écriture : son habilitation réelle sur une institution est relue en base à chaque appel via `resolve_institution_context`, jamais fait confiance depuis le JWT seul.

3.4 RLS « deux titulaires » et rôles PG applicatifs

Le financement institutionnel et le module Collateral reposent sur des *Row-Level Security policies* à deux titulaires (`company_tenant_id` / `institution_tenant_id`), pilotées par des GUC de session (variables PostgreSQL posées par l'application à la connexion, lues par les policies). **Quatre rôles PostgreSQL applicatifs** portent des privilèges distincts, déclarés dans `contracts/privilege-contract.yaml` :

RÔLE	PORTÉE
`tauraco_collateral_app`	Module Collateral + côté institution du branchement ERP→banque. Aucun privilège sur la comptabilité du client — une banque lit le snapshot, jamais les livres.
`tauraco_financing_app`	Parcours entreprise du financement institutionnel.
`tauraco_custody_app`	Le tiers détenteur (custodian) comme acteur à part entière (migration 081).
`tauraco_requeteur`	SELECT seul, allow-list de tables, `default_transaction_read_only=on`, `statement_timeout=5s` — moteur du requêteur conversationnel (§3.5).

`privilege-contract.yaml` déclare, table par table, ce que chaque rôle a le droit de faire (`select` / `insert` / `update` / `delete`) et pourquoi. `backend/tests/test_contrat_privileges_postgres.py` confronte ce contrat aux GRANT réels. Deux échecs, asymétriques :

- `MISSING_GRANT` — un privilège déclaré et absent en base (une migration n'a pas tourné, ou un droit a été révoqué) ;
- `UNEXPECTED_WRITE_GRANT` — une écriture présente en base et non déclarée (accès élargi sans le dire). Une *lecture* non déclarée est signalée sans faire échouer le test ; une *écriture* non déclarée, si.

Deux défauts de production sont nés d'une migration qui n'accordait ses droits qu'à un seul des deux rôles applicatifs, sans que rien ne le signale — l'un a rendu quatre routes en 500, l'autre empêchait de désigner une banque. **Ce contrat ne remplace pas la RLS** : un privilège autorise à interroger une table, les policies décident des lignes — les deux échouent pour des raisons différentes et sont donc nécessaires ensemble.

> **Règle « agrégat inter-tenants »** (postmortem Lot 4, salle de marché institutionnelle). Un agrégat inter-banques lu sous `session_institution` (RLS) ne voit **que** les lignes de l'appelante — un agrégat « je suis seul en lice » ou vide silencieux, invisible sur SQLite où la RLS n'existe pas. Règle retenue : tout agrégat inter-tenants doit se lire sous un **rôle propriétaire** cantonné à la fonction d'agrégat, **et** être couvert par un test Postgres-RLS passant par la **route réelle**, sans override câblé en CI.

3.5 Requêteur conversationnel (`/v2/ai/requeteur`)

Accès lecture seule premium pour dirigeants de PME, isolé sur trois axes — tenant, rôle, type d'accès — appliqués côté serveur, jamais pilotés par le prompt. Code dans `backend/app/services/requeteur/` .

EXIGENCE	FICHIER
Pas de SQL généré par le LLM — catalogues d'outils fermés	`catalogs/*.py`
Portée tenant — `RequeteurContext.tenant_id` injecté côté serveur	`dispatcher.py` , `output_filter.py`
Rôle/type d'accès — dérivation exclusive, un seul module catalogue importé	`access_type.py` , `dispatcher.load_catalog`
DB lecture seule — rôle `tauraco_requeteur`	migrations 030 + 031
Allow-list de tables	`allowlist.py`
Paramètres de portée interdits côté modèle	`dispatcher.invoke_tool` (`FORBIDDEN_PARAMS`)
Anonymisation investisseur	`anonymize.py` , `output_filter.py`
Journal d'audit par appel	`dispatcher.py` + `services/audit_log_service.py`

Ordre des défenses dans `POST /v2/ai/requeteur` (`api/routes/ai_assistant.py`) :

1. Auth (JWT).
2. **Dérivation du type d'accès** — `derive_access_type()` refuse les rôles admin, les profils en conflit, l'absence de profil correspondant (403, avant tout contrôle de facturation).
3. **Paywall** — `premium.etat_tenant()` (prédicat unifié, voir ci-dessous) ; sinon 402.
4. Sélection du catalogue — un seul module Python importé, les deux autres non.
5. Boucle d'outils Anthropic, `REQUETEUR_MAX_ITERS` itérations max.
6. Chaque appel d'outil passe par le point de passage unique (`dispatcher.invoke_tool`) : injection du contexte, rejet des paramètres interdits, validation du schéma, exécution sur le moteur lecture-seule, filtrage de sortie, écriture d'une ligne d'audit.

Le catalogue investisseur (`catalogs/investor.py`) reste branché malgré le gel du marketplace investisseur (Lot 1) — c'est l'exception assumée du mode libre.

Le gate « existence/habilitation avant paywall »

Sur `/v2/ai/insights` (mêmes fichiers), le paywall se décide sur **deux axes** — tenant et institution — et l'ordre compte : pour une page institution (préfixe `financing-bank-`), `institution_id` est d'abord requis (422 sinon), puis l'habilitation de l'appelant sur cette institution est résolue en base via `resolve_institution_context` (jamais fait confiance depuis le JWT) — et **seulement ensuite** l'abonnement premium de l'institution est vérifié (402). Aucun rôle plateforme ne court-circuite cette branche : le JWT d'un admin ne dit rien sur l'abonnement d'une banque partenaire donnée. `resolve_institution_context` distingue quatre refus (espace financeur absent, tiers du registre absent, aucune habilitation validée, tiers sans qualification) — chacun porté par une `CollateralError` typée (`HabilitationRefusee` → 403, `RessourceIntrouvable` → 404 selon le cas) — de sorte que la réponse ne fuite jamais si l'institution existe au-delà de « vous n'agissez pas pour elle ». Le principe se généralise : vérifier l'existence/l'accès à une ressource avant de facturer son contenu, jamais l'inverse.

3.6 Invariants produit — jamais de taux, jamais de classement

- **§43/§68** — les conditions de tarification (taux, commission, tenor) viennent des banques partenaires, jamais de la plateforme. Le moteur de commission `pricing_islamic` a été retiré au Lot 1 (`docs/legal/2026-09-14-fermeture-consultation-investisseur.md`). Ne jamais réintroduire un calcul de taux côté plateforme.
- **§13** — aucun classement ni recommandation d'offres. Ces invariants ne vivent pas dans la machine à états Legal Review (qui ne gère que le corpus AUS 2010) mais sont portés par des docstrings dans `app/domains/financing/{offers,acceptance,bank,marketplace}.py` et des tests-gates (`backend/tests/financing/test_gate_branchement.py`). Un amendement se matérialise par un document de décision dans `docs/legal/` puis la mise à jour des docstrings/gates dans le lot qui implémente.

04 Contrats et gates CI

4.1 Le manifeste fonctionnel

`tools/release/build_functional_manifest.py` extrait une surface fonctionnelle déterministe (routes API, routes SPA, vues, formulaires, stores, gates, workers, jobs planifiés, adaptateurs, migrations) directement depuis un arbre Git — AST Python + analyse texte conservative, sans importer le code applicatif. Comparé au HEAD via

`--compare`, il détecte toute route, vue ou migration retirée ou modifiée sans autorisation.

> **Piège documenté dans le script lui-même** : sans `--tree`, la commande la plus naturelle (`--output` sans `--compare`) lit `BASELINE_SHA` — un commit historique figé — **pas** l'arbre de travail. Régénérer le manifeste officiel de cette façon fait silencieusement disparaître toute surface ajoutée depuis ce commit, sans erreur visible : le fichier reste valide, et un `--compare` ultérieur rapporte `missing=0` parce qu'il compare le manifeste réduit à l'arbre réduit. Toujours passer `--tree` en nommant l'arbre réellement livré lors d'une régénération de baseline. `backend/tests/collateral/test_gate_manifeste_complet.py` garde contre une génération partielle atteignant la baseline commitée, en comparant au système de fichiers plutôt qu'à un autre arbre Git.

Trois sections approuvées, distinctes, dans `contracts/functional-baseline.json` :

SECTION	USAGE
`approved_replacements`	Documente un déplacement d'une entrée vers une cible survivante (renommage, refactor de surface).
`approved_authorization_changes`	Documente un changement de rôles/permissions sur une route existante.
`approved_removals`	Documente une suppression pure — sans cible. Chaque entrée exige trois clés : `entry`, `reason`, `decision` (référence à une décision tracée) — une suppression n'a pas de cible, ce qui la rend honnête, c'est sa justification et sa référence, jamais une simple absence passée sous silence.

Quand et comment approuver un changement : quand la CI (job `functional-manifest`, `.github/workflows/ci.yml`) rapporte une suppression ou un déplacement de surface légitime, ajouter l'entrée correspondante dans la section appropriée du manifeste baseline plutôt que de contourner le gate. La régénération préserve par défaut les sections déjà approuvées (`--reset-approved` les efface explicitement — à n'utiliser que pour repartir d'un manifeste propre, jamais pour faire disparaître une approbation gênante).

4.2 Contrats de schéma

`contracts/schema-production-v<tête>.json` est l'instantané du schéma de production **à la révision Alembic courante** (actuellement `v086`) — à ne pas confondre avec `schema-baseline-v050.json`, la baseline greenfield **figée** de `bootstrap_db.py`, qui ne bouge jamais. Le contrat de production se périme à **chaque** migration déployée : le régénérer après chaque lot, sur une copie restaurée puis migrée (jamais un montage depuis les modèles — la production porte des objets historiques hors migrations) :

```
docker compose exec backend python scripts/schema_contract.py \
  --url "<url de la copie migrée>" --output /tmp/snap.json
docker compose exec backend cat /tmp/snap.json \
  > contracts/schema-production-v<nouvelle tête>.json
```

L'exercice de restauration mensuel (§5.1) compare la base restaurée à ce contrat. Un exercice qui échoue sur `schema` avec un contrat en retard d'une révision n'invalide pas la sauvegarde — il dit que cette étape a été oubliée. C'est exactement ce qui s'est produit une fois : le contrat en usage datait de la révision 050, invérifiable depuis vingt-six révisions sans que personne ne le sache. Un dump frais restauré, migré, et confronté au contrat est un point fixe : c'est l'exercice mensuel qui le prouve, pas la seule existence du dump.

4.3 Les workflows CI et leurs pièges connus

WORKFLOW	DÉCLENCHEUR	RÔLE
`backend-ci.yml`	push (hors main, paths `backend/**`) + PR vers main	Smoke Stripe, vérifications de concurrence PostgreSQL (quota FREE atomique, locking checkout Stripe, rejeu mobile money, invitation expert), chaîne Alembic linéaire, migration fraîche jusqu'à tête, isolation Collateral, invariants Legal Review, isolation financement, rollback migrations Collateral
`ci.yml`	push + PR	« Functional contract CI » — manifeste fonctionnel complet, repositionnement de surface publique, sémantique du drift gate de déploiement, revendications publiques tauraco.ai/tauraco.io
`frontend-app.yml`	push + PR	Unit (Vitest), build (Vite), e2e Playwright contre un backend réel (Postgres + Redis + FastAPI) + contrats de formulaires déterministes
`mobile-ci.yml`	PR (paths `mobile/**`)	Matrice `tauraco-erp` / `tauraco-financement` : `npm ci`, typecheck, lint, tests
`protect-main.yml`	push + PR	Bloque tout push direct non-owner sur main ; exige les checks verts

Pièges connus, avec leur cause exacte :

- **Liste d'isolation via processus séparé** — la fixture `engine` de `tests/conftest.py` réécrit `Base.metadata` en place pour SQLite (`UUID → String(36)`) sans jamais l'annuler ; le mapper ORM mémoïse alors ce type figé pour tout le process. Les modules qui écrivent en SQL brut (la majorité des 296+ tests Collateral) y échappent ; ceux qui passent par l'ORM sur un vrai PostgreSQL (ex. `test_projection_persistence_postgres.py`, `test_role_reel_request_financing_postgres.py`) ne le peuvent pas — PostgreSQL refuse alors « column "id" is of type uuid but expression is of type character varying ». Solution : ces fichiers sont explicitement exclus du run groupé (`--ignore`) et exécutés dans une étape CI dédiée, à part.
- **Compteur de tables Collateral (37)** — le job « Verify Collateral isolation gates » affirme `assert tables == 37` sur `information_schema.tables` filtré `collateral%`. Toute migration qui ajoute (ou retire) une table `collateral_*` doit faire évoluer ce chiffre dans le même commit, sous peine de faire échouer la CI sans rapport apparent avec le changement.
- **Clés JWT jetables en CI** — le job e2e de `frontend-app.yml` génère une paire RS256 jetable (`openssl genpkey`) au début du job, jamais commitée ni journalisée : le runner GitHub n'a ni `secrets/` ni `/run/secrets`, et sans clé, `load_jwt_keys()` ferait échouer le boot d'uvicorn (voir §3.1 — aucun repli HS256 silencieux, y compris en CI).
- **Rôles applicatifs absents avant migration 053** — les tests d'isolation financement qui interrogent `has_table_privilege` pour les deux rôles applicatifs sont ignorés sur la base générique de test : ces rôles (et les tables `financing_*`) n'existent qu'à partir de la migration 053, jamais dans un schéma monté par `create_all_tables()`/SQLAlchemy.
- **`mobile-ci.yml`** ne se déclenche que sur PR touchant `mobile/**` — un changement backend qui casse un contrat consommé par les apps mobiles ne sera pas détecté par ce workflow.

05 Exploitation

5.1 Sauvegardes

Procédure unique : `scripts/backup_restore.py`, invoquée via le service compose `ops-backup` (`docs/runbooks/backup-restore.md`). Un `pg_dump | gzip` nu n'en fait

pas partie — il ne produit ni manifeste, ni empreinte, ni recensement du coffre de preuves, et une restauration à partir d'un tel dump ne peut pas être vérifiée.

GARANTIE	DÉTAIL
Dump `-Fc` avec propriétaires et privilèges	le cloisonnement RLS/GRANT survit à la restauration
Checksum	sha256 du dump dans `manifest.json` — la restauration **refuse** si l'empreinte diverge
Coffre de preuves	coffre injoignable ⇒ sauvegarde **interrompue** (fail-closed), pas une sauvegarde partielle silencieuse
Vérification post-restauration	schéma vs contrat, RLS/policies, preuves, décomptes
Trace `RESTORE_VERIFIED`	écrite dans `audit_logs` de la base d'exploitation, avec RPO/RTO observés

Sauvegarde quotidienne planifiée — cron hôte à **2h43** :

```
43 2 * * * almalinux /opt/tauraco/scripts/run_backup.sh >> /var/log/tauraco-backup.log 2>&
```

Chaque nuit, un sous-répertoire daté `quotidiennes/AAAAMMJJ-HHMMSS/` (dump + manifeste) est écrit dans le volume `backups`, puis la rétention s'applique : `BACKUP_RETENTION_JOURS`, défaut **30 jours**, bornée aux sous-répertoires datés (les snapshots `pre-<revision>` pris avant une migration ne sont pas couverts — ils se retirent à la main).

Exercice de restauration mensuel — un dump n'est pas une sauvegarde tant qu'on n'a pas restauré. L'exercice restaure le dernier dump dans une base jetable, vérifie contre `contracts/schema-production-v086.json`, et trace le résultat dans la base d'exploitation. Le préflight (`app/domains/collateral/exploitation/preflight.py`) déclare la trace périmée au-delà de `BACKUP_EXERCICE_MAX_JOURS` (défaut 90 jours) — d'où la cadence mensuelle : deux exercices manqués avant que le gate rougisse. Ce que la procédure ne fournit pas : point-in-time recovery (`point_in_time_recovery: false` dans le manifeste — c'est un instantané transactionnel) et export hors site du dump (le dump vit sur le disque de l'hôte ; seul le coffre de preuves R2 est répliqué par son fournisseur).

Cibles SLO (`docs/observabilite-slo.md`) : RPO ≤ 26h, RTO ≤ 30 min (dernier exercice observé : ~4 min), fraîcheur d'exercice ≤ 90 jours.

5.2 Observabilité et alertes

Le moteur d'alertes Collateral (`app/domains/collateral/alertes/`) distingue **23 types d'alerte** (`CollateralTypeAlerte`, `app/domain/collateral_enums.py`) — chiffre vérifié contre la contrainte CHECK `alerte_type_connu` posée par la migration 082, qui l'énumère à l'identique. Trois sévérités (`CollateralSeveriteAlerte` : `INFO`, `AVERTISSEMENT`, `CRITIQUE`) et quatre statuts (`OPEN`, `ACKNOWLEDGED`, `RESOLVED`, `SUPPRESSED`).

> L'historique de ce vocabulaire porte une leçon opérationnelle directe : la migration 075 a dû réparer la contrainte PostgreSQL après qu'elle a refusé des types d'alerte déjà déclarés côté Python (dix valeurs sur vingt exclues par la contrainte). La migration 077 a documenté ce qui arrive quand on n'élargit qu'une des deux tables concernées ; la 082 pose les **deux** contraintes à la fois (`alerte_type_connu` et `parametre_type_connu`) pour cette raison précise. **Toute extension du vocabulaire d'alerte doit mettre à jour l'enum Python ET les deux contraintes CHECK dans la même migration.**

Le passage quotidien du moteur d'alertes tourne en cron (`tauraco-collateral-alertes`, journal `/var/log/tauraco-collateral-alertes.log`), idempotent —

rejouable via `scripts/run_collateral_alertes.sh`. SLI/SLO mesurés via `GET /metrics` (Prometheus) : disponibilité des 4 surfaces (cible 99,5 %/mois), latence p95 lectures authentifiées (< 500 ms/jour), taux de 5xx (< 1 %/jour), retard de file soutenu (< 100 messages / 15 min). **Pas de Prometheus/Alertmanager versionné à ce jour** — `/metrics` est prêt à être collecté (scrape 15s recommandé) mais aucune règle d'alerte infrastructure n'est branchée : une violation de SLO se constate en la mesurant, elle ne réveille personne automatiquement. `scripts/verify_slo.sh` est rejouable à la main ou en cron pour vérifier surfaces, `/metrics`, planifications installées, âge du dernier dump et de la trace `RESTORE_VERIFIED`.

5.3 Seeds démo — inventaire et doubles gardes

SCRIPT	GARDE
`seed_financing_demo.py`, `seed_collateral_demo.py`	Garde simple — refusent hors `ENV=demo|staging`. Écrivent largement ; une exécution accidentelle serait grave.
`seed_production_demo_tenants.py`	**Double garde** — écrit le réseau démo pan-OHADA (17 États) *en production applicative*, dans le pé strictement délimité `environment_class=DEMO` (révision 073). Exige simultanément : `ALLOW_PRODUCTION_DEMO_SEED=true python -m scripts.seed_production_demo_tenants --allow-pr` Les deux sont nécessaires — un drapeau seul se recopie d'un historique de shell, une variable seu un environnement oublié. Ne fait **jamais** de `TRUNCATE` / `DELETE` global/ `DROP` — uniquement des cr idempotentes. Sans les deux gardes, l'outil décrit ce qu'il ferait et ne touche à rien.
`seed_tenant_naf_codes.py`	**Geste d'opérateur, jamais d'agent** — dépeuple/peuple `Tenant.naf_code` (invariant produit pour l' investisseur du requêteur, §3.5). Ne jamais l'invoquer de sa propre initiative.
`seed_collateral_referentiel.py`, `import_ohada_country_reference.py`, `charger_pack_legal_review.py`, `habiliter_approbateur_juridique.py`	Chargements référentiels/juridiques — `charger_pack_legal_review.py` exige `exiger_approver` ; i **aucune API** pour accorder une habilitation juridique — c'est une écriture en base délibérément m dans `docs/runbook.md`, même patron que `seed_tenant_naf_codes.py`.

5.4 Runbooks existants (table de renvoi)

RUNBOOK	SUJET
`docs/runbooks/backup-restore.md`	Sauvegarde, restauration, checksum (détaillé §5.1)
`docs/runbooks/deploy-frontend-app.md`	Déploiement `app.tauraco.io`, alias, drift gate, rollback (détaillé §1.4)
`docs/runbooks/institution-kyb.md`	Dossier KYB d'une institution financière partenaire
`docs/runbooks/onboard-financial-institution.md`, `create-partner-bank.md`	Intégration d'une nouvelle banque partenaire
`docs/runbooks/first-financing-request.md`, `facility-drawdown-servicing.md`	Cycle de vie d'une demande de financement, tirage, servicing
`docs/runbooks/collateral-incidents.md`, `collateral-premier-dossier.md`	Incidents et premier dossier Collateral
`docs/runbooks/legal-rule-validation-and-activation.md`	Machine à états de revue juridique
`docs/runbooks/demo-bout-en-bout-sega.md`, `demo-financial-institution.md`, `guide-scenarios-demo.md`	Parcours de démonstration complets
`docs/runbooks/add-update-country-overlay.md`	Ajout/mise à jour d'un overlay pays OHADA
`docs/runbooks/inventory-stock-finance-demo-validation.md`	Validation du sous-ledger de stock (financement sur stock)
`docs/runbooks/guide-captures.md`	Production de captures d'écran pour la documentation utilisateur
`docs/observabilite-slo.md`	SLI/SLO, actions en cas de violation (détaillé §5.2)
`docs/financing/surfaces-gelees.md`	Inventaire du code/tables gelés par le Lot 1 (marketplace investisseur)

5.5 Problèmes connus et solutions

SYMPTÔME	CAUSE PROBABLE	ACTION
Le backend ne démarre pas en production	`validate_production_security()` a levé `RuntimeError` — clés RS256 manquantes/malformées, `DEV_AUTH_ENABLED` / `TURNSTILE_QA_BYPASS_ENABLED` / `JWT_ACCEPT_LEGACY_HS256` à vrai, ou `CORS_ALLOW_ORIGINS` qui ne correspond pas exactement à l'allow-list	Lire le message d'erreur au boot (il no le réglage fautif) ; vérifier `secrets/jwt_private.pem` / `jwt_publi` montés en lecture seule
Une variable d'environnement modifiée dans `.env` ne prend pas effet	`docker compose up -d` seul ne recrée pas un service déjà démarré si sa définition n'a pas changé	`docker compose up -d --force-recrea <service>` (puis `docker compose rest nginx` si le service est `backend`, §1.3)
Un worker devient bruyant / boucle serrée sur erreur Redis	Connexion Redis perdue — le worker retente indéfiniment avec un `sleep` court (`ConnectionError`)	Vérifier `redis-cli -u $REDIS_URL pin` s'assurer d'un `socket_timeout` raisonnable côté client Redis pour évi les boucles de reconnexion trop rapprochées
CORS en production	Ne jamais élargir `CORS_ALLOW_ORIGINS` au-delà de l'allow-list — `validate_production_security()` refuse le boot si l'ensemble diverge, y compris par ajout	Toute nouvelle origine légitime doit d' être ajoutée à l'ensemble `expected` d `core/security.py`
« View-as » (vue-en-tant-que) expire après 15 min	Comportement voulu — jeton limité dans le temps, pas un bug de session	Relancer le view-as depuis l'interface d'administration
DB PostgreSQL inaccessible	Conteneur arrêté ou `DATABASE_URL` erroné	`docker compose ps postgres` → `dock compose restart postgres` ; vérifier le logs
Redis inaccessible	Conteneur arrêté	`docker compose restart redis` — les workers se reconnectent automatiquement
Upload de facture en échec (500 sur `/v2/invoices/init-upload`)	R2/MinIO inaccessible	Vérifier les identifiants R2 (prod) ou redémarrer MinIO (dev) ; les factures cours restent en `UPLOADED` jusqu'au r du stockage

06 Mobile

Deux applications Expo/React Native distinctes sous `mobile/` : **`tauraco-erp`** (comptabilité OHADA, facturation, trésorerie, scanner OCR) et **`tauraco-financement`** (portail vendeur — mes factures, mes financements, mes contrats, portefeuille, KYB). Chacune a son propre `eas.json`, ses propres identifiants de package (`io.tauraco.erp` / `io.tauraco.financement`), et ses propres credentials.

6.1 Auth — refresh dans le corps, rotation

Chaque app rafraîchit son jeton toutes les 14 minutes (`hooks/useRefresh.ts`, `REFRESH_INTERVAL_MS`) — sous le TTL de 15 min de l'access token — et à chaque retour au premier plan (`AppState`). Le refresh token est lu depuis `SecureStore` par un chemin **mutexé et dédupliqué** partagé avec l'intercepteur axios 401 (`services/api.ts`) : utiliser directement l'état Zustand local rejouerait un token que l'intercepteur vient peut-être de faire tourner. Le serveur suit exactement le mécanisme du §3.2 : `refresh_token` dans le corps JSON, rotation à chaque appel, révocation de toute la famille en cas de rejeu détecté.

6.2 Build et soumission EAS

```
# Build de développement (modules natifs)
eas build --profile development --platform android|ios

# Production
eas build --profile preview --platform android      # APK test interne
eas build --profile production --platform android   # AAB Google Play
eas build --profile production --platform ios        # IPA

# Soumission
eas submit --platform android --profile production
eas submit --platform ios --profile production

# Mise à jour OTA (JS seul, pas de module natif)
eas update --branch production --message "..."
```

6.3 Credentials — emplacement, par nom seulement

CREDENTIAL	FICHIER (NOM)	RÉFÉRENCÉ PAR
Compte de service Play Console	`mobile/store/play-service-account.json` (non suivi par git — renommé depuis `google-services.json` pour éviter la confusion avec un vrai fichier Firebase)	`serviceAccountKeyPath` dans les deux `eas.json`, `mobile/store/deploy_play_store.py`, `check_play_access.py`
Clé App Store Connect	`AuthKey_8TDK96JUWW.p8`	`ascApiKeyPath` dans les deux `eas.json`

Clé orpheline à révoquer : `mobile/ApiKey_V6LH7WEV7KXI.p8` n'est référencée nulle part dans le dépôt (ni dans les `eas.json`, seule `AuthKey_8TDK96JUWW.p8` est utilisée). Elle ne doit être ni committée ni redistribuée — à révoquer côté App Store Connect (Users and Access → Integrations) puis supprimer le fichier localement une fois la révocation confirmée.

6.4 Le piège `expo prebuild --clean`

`android/` et `ios/` sont **gitignorés** et intégralement **régénérés** à chaque `expo prebuild --clean`. Conséquences concrètes, à refaire après chaque prebuild :

- `android/local.properties` disparaît — le recréer (`echo "sdk.dir=$ANDROID_HOME" > android/local.properties`) avant tout build Gradle.
- Toute édition manuelle d'`AndroidManifest.xml` (ex. `android:usesCleartextTraffic="true"` pour tester contre un backend local en HTTP, jamais en production où l'API est en HTTPS) est perdue et doit être reportée.
- Si un plugin Firebase est un jour ajouté à `app.json`, `expo prebuild` peut recopier un `google-services.json` à la racine de chaque app — piège actuellement inerte (aucun `app.json` ne référence `googleServicesFile`) mais à surveiller si Firebase est introduit.

`android/app/build.gradle` fait pointer `signingConfigs.release` vers le même `signingConfigs.debug` que `debug` — `assembleRelease` fonctionne donc sans keystore réel pour les besoins de test local, mais ce n'est **pas** la configuration de signature utilisée par les builds EAS de production.

6.5 État de publication

Les deux applications disposent d'une configuration EAS complète (build + submit), de textes et captures store (`mobile/store/<app>/`), et d'un runbook de vérification émulateur rejouable (`mobile/scripts/emulator-verify.md`) qui déroule un parcours complet contre le backend réel avant toute publication. Toute capture ajoutée aux fiches store doit être relue visuellement (au moins 4 par app) avant commit : données de démo uniquement, pas d'heure ou de batterie incongrue.

07 Évoluer sans casser

7.1 Workflow attendu

1. Lire le code existant concerné, proposer un plan avant tout changement (domaine/auth : tests pytest écrits **avant** l'implémentation).
2. Une branche par changement — jamais de commit direct sur `main` (`protect-main.yml` bloque le push direct des non-owners). Nommage : `feat/`, `fix/`, `docs/`, `chore/`, `refactor/`, `test/<description-courte>`.
3. Commits au format Conventional Commits (`<type>(<scope>): <message>`, anglais, présent, sans point final).
4. Les quatre workflows CI (backend-ci, ci — manifeste fonctionnel, frontend-app, mobile-ci si `mobile/**` touché) doivent passer, plus `protect-main.yml` qui attend le smoke check backend sur la tête de la PR.
5. Au moins un reviewer (@segadiarrah) avant merge, squash sur `main`.
6. Déploiement — backend : migrer puis `docker compose restart backend` (§2.3, §5) ; SPA : `deploy_app_canonical.sh` + `deploy_app_alias.sh` depuis le même commit + `check_frontend_drift.sh` (§1.4).

7.2 Ajouter un outil au requêteur conversationnel

1. Ajouter un schéma d'entrée Pydantic avec `model_config = ConfigDict(extra="forbid")` dans le module catalogue concerné. **Ne jamais** inclure `tenant_id`, `user_id`, `investor_id` ou tout champ de portée — ils sont injectés depuis `ctx`.
2. Écrire un handler `(ctx, params, ro_db) -> list[dict]` qui filtre par `ctx.tenant_id` (ou `ctx.self_investor_id` pour le catalogue investisseur) et ajoute une `LIMIT` explicite.
3. Ajouter un `ToolSpec` à `TOOLS_SPECS` du module, avec descriptions FR et EN.
4. Si le handler lit une nouvelle table, l'ajouter à `allowlist.py` **et** livrer une nouvelle migration Alembic qui `GRANT` le SELECT à `tauraco_requeteur`.
5. Ajouter un test dans `tests/requeteur/` — scénario d'isolation ou vérification d'anonymisation. Les 25 tests d'isolation existants (`docker compose exec backend pytest tests/requeteur/ -v`) doivent tous rester verts.

7.3 Ajouter une migration

Checklist — les trois éléments suivants se livrent **dans la même tâche**, jamais en suivi séparé :

1. Migration idempotente numérotée en séquence (§2.3), jamais de modification d'une migration déjà appliquée.
2. Bump de `TETE_ATTENDUE` dans les fichiers de test concernés (`test_institutional_baseline.py`, `test_migration_harness.py`, `test_production_baseline_v2.py`) — sinon la CI échoue sur une référence désynchronisée.
3. Régénération du manifeste fonctionnel si la migration ajoute/retire une surface qu'il suit (tables listées dans `ARRAY_CATEGORIES`, §4.1).

Après déploiement en production, régénérer `contracts/schema-production-v<nouvelle tête>.json` (§4.2) — sans quoi l'exercice de restauration mensuel échoue silencieusement sur un contrat périmé.

Si la migration touche une table `collateral_*` : mettre à jour le compteur figé à 37 dans `backend-ci.yml` (§4.3). Si elle étend le vocabulaire d'alerte : mettre à jour l'enum Python **et** les deux contraintes CHECK (`alerte_type_connu`, `parametre_type_connu`) dans la même migration (§5.2).

7.4 Ajouter une route

modèle ORM → enum → migration Alembic → schéma Pydantic (requête + réponse) → service (`services/<nom>_service.py`) → route (`Depends(require_roles(...))`) → tag OpenAPI cohérent → inclusion du router dans `main.py` si nouveau fichier → tests (chemin heureux + rejet) → mise à jour du manifeste fonctionnel si la surface est suivie (§4.1) — sinon la CI « Complete functional manifest » rapportera une addition non approuvée à documenter dans `approved_replacements` / `approved_removals` selon le cas.

7.5 Pièges Vue payés (frontend-app)

- **Prop booléenne optionnelle sans `default`** — une prop `boolean` optionnelle doit avoir un défaut explicite `false`, **jamais** `undefined`. Un défaut manquant a laissé 25 `PremiumGate` verrouillés en environnement réel (le composant traitait `undefined` différemment de `false` dans sa condition d'affichage). `withDefaults` avec une valeur `undefined` exige de vérifier la spec réelle du composant consommateur, jamais une supposition.
- **`gen:api` toujours depuis un export OpenAPI frais** — régénérer `src/api/generated/schema.d.ts` contre un uvicorn dont le code a changé sans redémarrage (§1.3, pas de hot-reload) régénère un schéma **périmé** qui compile mais ment sur la forme réelle des réponses. Toujours `docker compose restart backend` avant `npm run gen:api` après un changement de schéma de réponse.
- **Convention UI — colonne Actions toujours en dernière position** dans les tableaux PrimeVue ; un dialog de révision doit être pré-rempli de la *saisie antérieure* de l'utilisateur (banque, vendeur…), jamais présenté comme une suggestion du système — la distinction entre « ce que vous avez déjà saisi » et « ce que le système propose » est significative pour l'utilisateur final.