

Doctrine

- [Conventions de code](#)
- [Tests](#)

Conventions de code

Conventions de code

📅 Dernière mise à jour : 2026-05-10

Source de vérité : `CONTRIBUTING.md`. Cette page reformule les règles structurantes.

Règles d'or backend

1. Logger central uniquement

- Tout log passe par `src/lib/logger.ts`
- Aucun `console.*` dans `src/` (sauf `src/config.ts` qui boot avant le logger)
- Niveau filtré via `LOG_LEVEL` (`debug` / `info` / `warn` / `error`)
- JSON ligne en prod, pretty en dev
- Préfixer le scope : `logger.info('[meta-sync] ...')`

Vérifier :

```
grep -rn "console\." src/ --include="*.ts" | grep -v src/lib/logger.ts | grep -v src/config.ts  
# → doit retourner zéro résultat
```

2. Variables d'env centralisées

- Toute env var déclarée dans `envSchema` de `src/config.ts` (Zod + défaut + commentaire)
- Lue uniquement via `import { config } from '<path>/config.js'` + `config.MA_VAR`
- Drizzle Kit (`drizzle.config.ts`) est la seule autre exception (s'exécute hors compile TS)

Quand on ajoute une var, **3 fichiers en parallèle** :

1. `src/config.ts`

2. `.env.example` (commentaire explicite)
3. `unraid/boardgame-referee.xml`

Et si la var change un comportement structurel, mettre à jour `ARCHITECTURE.md`.

3. Helper `envBool()` pour les booléens

- `z.coerce.boolean()` → `Boolean("false") === true` en JS (piège `VISION_ENABLED=false` silencieusement `true`)
- Utiliser `envBool(defaultValue)` défini en haut de `src/config.ts`. Comprend `true/false/1/0/yes/no/on/off`.

4. Repository pattern (Phase 2)

- **Seuls** les fichiers `src/repositories/*.repo.ts` ont le droit d'importer `db`, `drizzle-orm` ou les tables du schéma
- Routes, services, middleware, scripts, crons : passent **toujours** par un repository
- Nommer les fonctions par intention métier (`getByBggId`, `setIngestStatus`), pas par verbe Drizzle (`select1`)
- Si une route a besoin d'une nouvelle requête : l'ajouter au repository

5. Services découpés (Phase 3) — barrel imports

- Un dossier par gros service (`services/rag/`, `services/ingest/`, `services/chunking/`)
- Toujours importer depuis `index.ts` (le barrel) du dossier, jamais un sous-module direct
- Le barrel est le contrat public, le reste est implémentation
- Aucun fichier > 350 lignes — découper en `types.ts` + un fichier par responsabilité

6. Stages d'ingestion : signature standard

```
async function runStageX(  
  game: Game,  
  ...,  
  push: EventPusher  
) : Promise<...>
```

- `push('event_name', payload)` pour l'UI (jamais `console.log`)

- Logger `info` pour le diagnostic serveur
- Erreurs non-fatales : try/catch local ou autour de l'appel dans le coordinator
- Erreurs fatales : remontent au try global de `coordinator.ts`

7. Aucun cache global dans `services/rag/retrieve/`

- Les structures partagées entre étapes (`seen: Set<string>`, `candidateRrfScores: Map<string, number>`, `synergyCards/metaCards`) sont créées dans `orchestrator.ts` et passées en paramètre aux 5 étapes — qui les MUTENT explicitement
- Aucun cache, singleton, ou module-scope state
- Raison : si une étape crée sa copie locale (par closure ou import circulaire), les passes redécouvrent les mêmes chunks → doublons en sortie ou blending RRF qui s'éteint silencieusement

8. Code partagé — ne pas réinventer

Avant d'écrire un utilitaire, vérifier qu'il n'existe pas dans :

- `slugify(name)` : `src/lib/utills.ts`. Une divergence casserait le lien nom-jeu / collection-Qdrant / nom-fichier-PDF.
- **Schémas Zod transverses** : `src/lib/schemas.ts` (`askRetrieveSchema`, `askStreamSchema`, `askFeedbackSchema`, `adminFeedbackQuerySchema`, `gameIngestMetadataSchema`, `decksParseSchema`) + constantes `CONTENT_TYPES`, `RULES_LANGUAGES`. Schéma strictement local (usage unique) peut rester dans la route.
- **Flux SSE frontend** : `useEventStream` (`composables/useEventStream.ts`). Ne jamais réécrire `fetch().getReader()` + parsing SSE manuel. Pour un endpoint métier, wrapper `useEventStream` dans un composable dédié (cf. `useAskStream.ts`).
- **Heartbeat des streams longs (backend)** : `streamSSE` qui peut rester silencieux >30s ouvre `setInterval` 8s avec `{ type: 'heartbeat' }`. Clear dans `finally`.
- **Fallback streams longs** : si la réponse est persistée en DB avant la fin du stream, exposer `GET /api/<resource>/:id` retournant 200/202/404. Front démarre polling si SSE casse + `meta.questionId` connu.

Règles d'or frontend

1. Pas de scoped CSS qui contredit Tailwind

- `display: flex` scoped écrase `hidden lg:flex` du template
- Préférer Tailwind utilities partout. `@apply` si tu dois absolument grouper.

2. Type-check via `--build` (pas `--noEmit`)

- La CI fait `npm run build` (`vue-tsc --build + vite build`)
- `--noEmit` peut passer en local et fail en CI
- Toujours `npm run build` avant de push

3. Regex avec flag `u` pour Unicode

- `\b` sans flag `u` rate les boundaries Unicode (é, ñ, accentués)
- Lookahead Unicode-aware avec flag `u` : `(?=\s|$|[\p{L}\p{N}_])`
- `normalizeCardKey()` (`useArbiterMarkdown.ts:74-81`) : NFC + lowercase + apostrophe/tiret normalization

4. Composants async pour les modales lourdes

```
const CardZoomModal = defineAsyncComponent(() => import('./components/CardZoomModal.vue'))
```

CardZoomModal + DeckImportModal : ~50 KB gzip retirés du bundle initial PlayView.

Conventions de nommage

- **Fichiers backend** : `kebab-case.ts` (`claude-ssh.ts`, `validate-model.ts`)
- **Fichiers frontend** : `PascalCase.vue` pour composants, `camelCase.ts` pour composables (`useAskStream.ts`)
- **Stores Pinia** : `useFooStore`
- **Composables** : `useFoo`
- **Types** : `PascalCase` (`SessionUser`, `RetrievedChunk`)
- **Constantes** : `UPPER_SNAKE_CASE` (`STICKY_WITH_DECK_CAP`, `MAX_INJECTED_IMAGES`)

Convention de commits

Convention courte façon Conventional Commits :

```
type(scope): description courte
```

Optional body

Exemples actuels :

- `security(ssh): isolate Claude SSH on dedicated 'oracle' user`
- `perf(rag): parallelize Qdrant searches + adopt setTimeout (8 sites)`
- `refactor(frontend): split AdminFeedbackView – Detail + helper markdown`
- `feat(ingest): OCR auto pour PDFs scannés (tesseract)`

Types courants : `feat`, `fix`, `refactor`, `perf`, `chore`, `docs`, `test`, `style`, `security`.

Linter / formatter

Pas de eslint/prettier formel actuellement. Convention manuelle :

- Indentation 2 espaces
- Pas de `;` semi-colons obligatoires (TS le tolère mais on en met)
- Quotes : single `'` partout sauf JSX (qui utilise double `"`)

Si tu veux ajouter eslint+prettier : config standard Vue + TS, no-config-thrasher.

Tests

Tests

“ Dernière mise à jour : 2026-05-10

Deux types de tests, complémentaires :

1. **Vitest unitaires** : déterministes, détectent les régressions silencieuses
2. **Banc d'éval RAG** : qualité subjective de l'oracle (judge Haiku)

Vitest unitaires

Périmètre

“ **Toute fonction critique non triviale ajoutée à** `src/lib/`, `src/services/`, `frontend/src/composables/` **ou** `frontend/src/stores/` **doit avoir un test unitaire** dans le même dossier sous la forme `*.test.ts`.

Les routes et les vues n'en ont pas besoin (pure orchestration), mais leurs services sous-jacents si.

Garde-fous prioritaires

- Schémas Zod (validation API)
- Parsers (markdown, scrapers HTML, decklists)
- Normalisations (clés de cache, slugs, normalize card key)
- Wrappers d'I/O (timeout, retry)
- Stores qui persistent en localStorage

Mock le strict minimum

```
// OK – mock config + repos quand on touche fs / DB
vi.mock('../config.js', () => ({ config: { LOG_LEVEL: 'debug', ... } })))
vi.mock('../repositories/games.repo.js')

// PAS OK – pas de mock fs/path
// → utiliser os.tmpdir() + randomUUID() pour un vrai dossier isolé
```

Plus simple, moins fragile.

Lancer

```
# Backend
npm test           # Run complet
npm run test:watch # Watch

# Frontend
cd frontend && npm test
```

CI

`npm test` doit **passer** en CI (job `test` dans `.gitea/workflows/build.yml`). Si un test échoue, on corrige le code (ou le test si l'attendu était faux). On ne skip pas.

Banc d'éval RAG

Différence avec les tests unitaires

	Vitest unit	Banc RAG
But	Régressions silencieuses dans la mécanique	Qualité subjective de l'oracle
Source	Code TS	Questions + ground truth
Verdict	Déterministe (assert)	Subjectif (judge LLM)

	Vitest unit	Banc RAG
Coût	Gratuit	Quota Claude (Haiku)
Fréquence	À chaque commit	Manuellement, après gros changements RAG

Localisation

`tests/rag/` — séparé de Vitest, lancé avec ses propres scripts.

Commandes

```
npm run test:rag          # Run complet
npm run test:rag:capture  # Capture les réponses pour archive (snapshot)
npm run test:rag:report   # Rapport markdown
npm run test:rag:html     # HTML interactif
npm run test:rag:html:serve # Serveur local viewer (port 8888 défaut)
```

Workflow typique

1. Faire un changement RAG significatif (HyDE, RRF, prompt, retrieval params...)
2. `npm run test:rag` → run les questions test
3. `npm run test:rag:html` → générer rapport HTML interactif
4. `npm run test:rag:html:serve` → ouvrir <http://localhost:8888> et comparer les réponses
5. Décider : régression acceptable / fix nécessaire

Alimenter le banc

- Source initiale : questions choisies par toi qui couvrent les cas critiques (rules / synergy / meta / deckbuilding par TCG)
- Auto-feed : feedbacks votés `down` avec commentaire dans `/admin/feedback` deviennent des candidats naturels
- Manuel actuellement (pas d'extraction auto). Workflow propre :
 1. Sélectionner les feedbacks down avec commentaire
 2. Reformuler en cas-test
 3. Ajouter à `tests/rag/dataset.json`
 4. `npm run test:rag` pour mesurer la régression

Tests frontend

Vitest + Vue Test Utils. Localisé sous `frontend/src/**/*test.ts`.

Couvre :

- Composables (`useAskStream`, `useMentionAutocomplete`, `useArbiterMarkdown`, `useStickyMentions`, etc.)
- Stores Pinia (`auth`, `games`, `session`)
- Helpers (`mana.ts`, `fab-symbols.ts`, etc.)

Pas de tests E2E Playwright actuellement. Pourrait être ajouté plus tard pour les flows critiques (login → ingest jeu test → poser question → vérifier réponse).

Coverage

Pas de seuil enforced. Vitest peut générer un rapport `--coverage` :

```
npm test -- --coverage
```

Mais pas de fail-on-low-coverage. Convention : si tu touches une fonction critique, le test existe.

Pièges connus

Mock partiel d'un module ESM

- Vitest avec ESM peut se mélanger les pinceaux sur `vi.mock`. Toujours déclarer le mock **avant** l'import du module à tester.

Tests qui dépendent de la timezone

- Garder `process.env.TZ = 'Europe/Paris'` ou `'UTC'` constant
- Sinon des tests passent en local et fail en CI selon le serveur

Tests qui dépendent de fichiers réels

- Préférer `os.tmpdir() + randomUUID()` plutôt que `mock fs`
- Cleanup dans `afterEach (fs.rm(dir, { recursive: true }))`

Tests qui appellent vraiment l'API Claude

- ☐ Jamais en CI (coût + flake)
- Mock le service ou utiliser le mode `CLAUDE_USE_LOCAL=true` avec un binaire stub