Réponses théoriques (Étapes 4 à 7)

5) (Q4) Architecture de données & stockage

Objectif

En production, il faut :

- stocker les résultats d'analyse,
- garder l'historique des requêtes,
- cacher les données collectées,
- gérer les configurations des agents, tout en supportant la concurrence (plusieurs analyses simultanées) et l'auditabilité.

Schéma de données (proposition)

A) Historique des requêtes

Table queries

- query_id (UUID, PK)
- query_text (TEXT)
- created_at (TIMESTAMP)
- status (queued/running/succeeded/failed)
- params (JSONB) : sources, langue, options…
- fingerprint (TEXT) : hash de query_text normalisée + params (pour déduplication/caching)

B) Cache des données collectées

Table collected_products_cache (ou cache Redis, voir ci-dessous)

- cache_key (TEXT, PK)
- query_text (TEXT)
- fetched_at (TIMESTAMP)
- ttl_expires_at (TIMESTAMP)
- source (TEXT)
- payload (JSONB) : liste de produits bruts

C) Résultats d'analyse

Table analysis_results

- analysis_id (UUID, PK)

- query_id (FK)
- created_at (TIMESTAMP)
- market_summary (JSONB)
- sentiment_summary (JSONB)
- products_normalized (JSONB)
- report_path (TEXT) ou report_html (TEXT)
- tool_versions (JSONB) : versions/sha/prompt_version

D) Configuration des agents

Table agent_configs

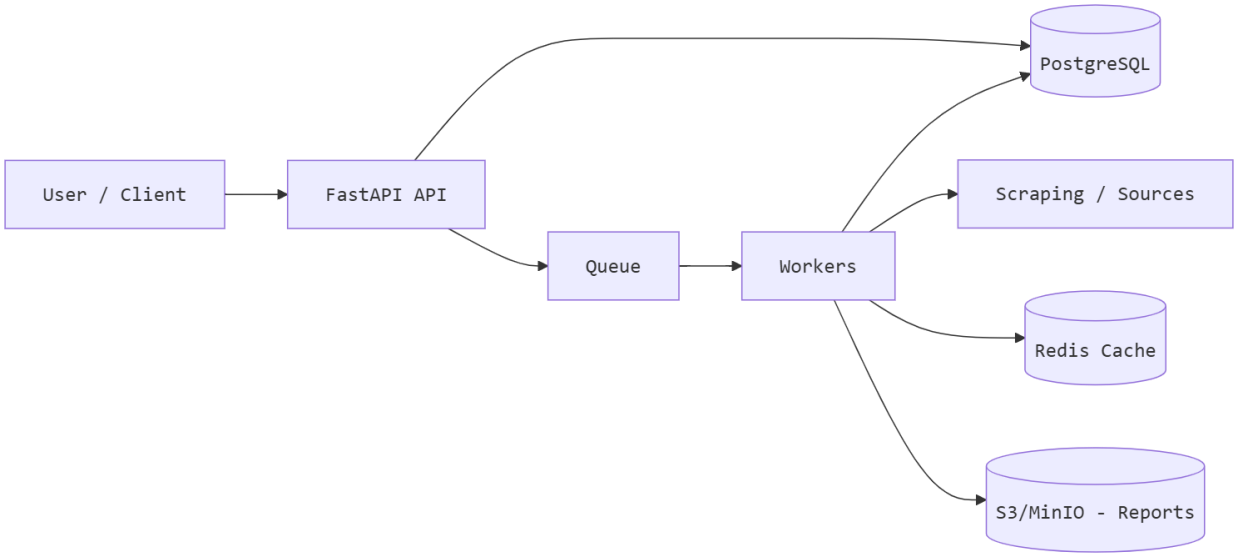
- config_id (UUID)
- name (TEXT)
- enabled_sources (JSONB/ARRAY)
- thresholds (JSONB)
- prompt_version (TEXT)
- updated_at (TIMESTAMP)

Système de stockage recommandé (et pourquoi)

- **PostgreSQL** comme “source of truth” :
    - solide pour l’historique, les relations, l’audit, et le stockage flexible via JSONB.
- **Redis** pour le cache (TTL natif, très rapide) :
    - cache des produits collectés,
    - cache des résultats d’analyse,
    - déduplication via fingerprint (évite recalculs).
- **Object Storage** (S3/MinIO) pour les rapports HTML si volumineux :
    - stockage scalable et peu coûteux, servi facilement.

Trade-off assumé :

- En démo/local, écrire des fichiers HTML suffit.
- En prod multi-instances, un stockage centralisé (S3/DB) devient nécessaire.



6) (Q5) Monitoring & observabilité

Tracing (traçage)

- Tracer une requête bout-en-bout (un “trace id” par analyse).
- Créer un span par étape :
  - fetch_products
  - analyze_market
  - analyze_sentiment
  - generate_report
- Ajouter des tags utiles :
  - query_id, cache_hit, num_products, source, error_type

Métriques de performance (exemples)

- Latence end-to-end : analysis_total_latency_ms (p50/p95/p99)
- Latence par étape : scraping, market, sentiment, report
- Taux d’erreur : analysis_failures_total
- Débit : analyses_completed_total
- Taille : products_count, report_size_bytes
- Cache : cache_hit_rate

Alerting

- Alerte si :

- taux d’erreur dépasse un seuil,
- p95 latence explose,
- backlog augmente (si exécution async via queue),
- scraping live échoue trop souvent (bascule excessive en fallback).

Mesurer la qualité des outputs

- Contrôles déterministes (sans LLM) :
  - validation de schéma (clés obligatoires),
  - valeurs NaN / incohérences,
  - sentiment_breakdown cohérent (somme / types).
- Score “qualité” (si LLM) :
  - un judge évalue cohérence, clarté, couverture.

7) (Q6) Scaling & optimisation

Gérer des pics de charge (100+ analyses simultanées)

Approche recommandée :

- API “front” rapide (FastAPI) qui :
  - enregistre la requête,
  - pousse un job dans une **queue**,
  - répond 202 + query_id.
- Workers “back” (autoscalables) qui exécutent :
  - scraping → analyses → génération rapport.

Technos possibles :

- Queue : Redis/RabbitMQ/Kafka
- Workers : Celery/RQ/Arq
- Déploiement : Docker + Kubernetes (HPA)

Optimiser les coûts LLM (si activé)

- Appeler le LLM en dernier (après filtres + stats).
- Réduire le contexte (top-K produits, résumé).
- Mettre en cache les réponses LLM :

    - clé = fingerprint + prompt_version + model.
- Utiliser un petit modèle pour tâches simples, un modèle plus fort uniquement si besoin.

Cache intelligent

- Cache multi-niveaux :
    1. produits collectés (TTL court, ex 1h),
    2. résultats d'analyse (TTL moyen, ex 24h),
    3. sorties LLM (TTL + versioning prompt/model).
- Invalidation :
    - par TTL,
    - par changement de config/prompt_version.

Paralléliser l'analyse

- Scraping multi-sources en parallèle.
- Market + sentiment en parallèle une fois les produits prêts.
- Rapport généré après agrégation.

8) (Q7) Amélioration continue & A/B testing

Évaluation automatique (LLM as Judge)

- Construire un set de requêtes "golden".
- Générer des rapports avec plusieurs versions.
- Noter automatiquement :
    - cohérence avec les chiffres,
    - couverture des points clés,
    - clarté,
    - absence d'invention (hallucinations).

Comparer des stratégies de prompt

- Versionner prompts (prompt_version).
- A/B testing :
    - assignation stable (hash de user_id ou fingerprint),
    - comparaison sur qualité, coût tokens, latence.

Feedback loop utilisateur

- Collecter une note (1–5) + commentaire + signal “utile/pas utile”.
- Relier le feedback à analysis_id.
- Exploiter le feedback pour ajuster filtres, règles, prompts.

Faire évoluer les capacités

- Ajouter de nouvelles sources via une interface “tool” stable.
- Normalisation plus robuste (devises, taxes, condition neuf/occasion).
- Déduplication produits (entity resolution).
- Ajout d’un mode “explainability” (justification de la recommandation).

9) Notes & limites actuelles

- Le scraping live peut être désactivé pour éviter blocages IP pendant la démo.
- Le sentiment est simulé (basé sur ratings) : en production, ingestion de vrais avis + modèle NLP/LLM + évaluation.
- La tendance 30 jours est simulée : en production, stockage réel des historiques de prix.