Q4 — Architecture de données & stockage (texte continu)

En production, je mettrais en place une architecture de données qui permet de conserver l'historique des requêtes, de stocker les résultats d'analyse, de mettre en cache les données collectées et de gérer les configurations des agents, tout en supportant plusieurs analyses en parallèle et en gardant une bonne auditabilité. Concrètement, j'utiliserais une base relationnelle (PostgreSQL) comme "source of truth", avec une table queries pour suivre chaque demande (texte, paramètres, statut, timestamps) et un champ fingerprint calculé à partir de la requête normalisée et des paramètres afin de faciliter la déduplication et le caching. Je prévoirais aussi un stockage pour les données collectées sous forme de cache (idéalement Redis pour le TTL et la vitesse, ou une table type collected_products_cache si on veut rester 100% SQL), afin d'éviter de re-scraper inutilement et de stabiliser la latence. Les résultats seraient enregistrés dans une table analysis_results reliée à queries, en stockant les résumés (market/sentiment), les produits normalisés, ainsi qu'un champ de traçabilité (versions des outils, commit SHA, prompt_version) pour pouvoir reproduire un résultat. Enfin, je garderais une table agent_configs pour versionner les réglages (sources activées, seuils, prompts), ce qui facilite les changements sans redeployer. Pour les rapports HTML, en local un simple fichier suffit, mais en production multi-instances je privilégierais un stockage objet (S3/MinIO) afin d'éviter de dépendre d'un disque local et pour servir des rapports plus volumineux de manière scalable.

Q5 — Monitoring & observabilité (texte continu)

Pour l'observabilité, je tracerais chaque analyse de bout en bout avec un identifiant unique (trace id) et je découperais le traitement en spans correspondant aux étapes principales, par exemple fetch_products, analyze_market, analyze_sentiment et generate_report. À chaque étape, j'ajouterais des tags simples mais utiles comme query_id, la source, le nombre de produits, un indicateur de cache_hit, et, en cas d'échec, un error_type clair. Côté métriques, je suivrais la latence globale (p50/p95/p99) et la latence par étape, le taux d'erreur, le débit d'analyses complétées, ainsi que des métriques "produit" comme la taille du rapport et le nombre d'items analysés, sans oublier un indicateur de performance du cache (hit rate). Pour l'alerting, je déclencherais des alertes si le taux d'erreur dépasse un seuil, si la p95 de latence dérive fortement, si un backlog augmente (dans le cas d'une exécution asynchrone via queue), ou si le scraping live échoue trop souvent et force un recours excessif au fallback. Enfin, pour mesurer la qualité des outputs, je commencerais par des contrôles déterministes (validation de schéma, cohérence des champs, absence de NaN, vérification que la distribution de sentiment "somme bien"), et si un LLM est utilisé, j'ajouterais une évaluation automatique de cohérence et de clarté via un "judge", surtout pour détecter les sorties incohérentes avec les chiffres.

Q6 — Scaling & optimisation (texte continu)

Pour absorber des pics de charge (par exemple 100+ analyses simultanées), je séparerais clairement l'API et l'exécution du travail. L'API FastAPI resterait "fine" : elle enregistre la requête, crée un query_id, pousse un job dans une queue et renvoie immédiatement une réponse 202 Accepted avec l'identifiant, afin d'éviter de bloquer des requêtes longues. Le traitement serait fait par des workers back-end autoscalables qui exécutent le scraping, l'analyse marché, l'analyse sentiment et la génération du rapport. Pour la queue, Redis/RabbitMQ/Kafka sont possibles selon le contexte, et côté workers on peut s'appuyer sur Celery/RQ/Arq, puis déployer avec Docker et éventuellement Kubernetes (HPA) pour scaler horizontalement. Si un LLM est activé, je limiterais les coûts en ne l'appelant qu'en dernier, après

avoir appliqué des règles et des stats qui réduisent le besoin de contexte, en ne passant que le top-K produits et un résumé, et surtout en mettant en cache les réponses avec une clé qui inclut le fingerprint, la prompt_version et le modèle. J'utiliserais aussi une stratégie "petit modèle par défaut / grand modèle seulement si nécessaire" pour les cas difficiles. Enfin, je mettrais en place un cache multi-niveaux (produits collectés avec TTL court, résultats d'analyse avec TTL moyen, sorties LLM avec TTL et versioning), et je paralléliserais ce qui peut l'être : multi-sources en parallèle au scraping, puis marché et sentiment en parallèle dès que les produits sont prêts, avant l'agrégation finale et la génération du rapport.

Q7 — Amélioration continue & A/B testing (texte continu)

Pour améliorer le système dans le temps, je mettrais en place une évaluation automatique sur un jeu de requêtes "golden" représentatif, afin de comparer plusieurs versions du pipeline et des prompts. L'idée est de générer des rapports avec différentes variantes, puis de mesurer de façon stable des critères comme la cohérence avec les chiffres, la couverture des points importants, la clarté, et surtout l'absence d'invention (hallucinations) lorsque des composants génératifs sont utilisés. Je versionnerais explicitement les prompts via un prompt_version et je ferais de l'A/B testing avec une assignation stable (par hash du user_id ou du fingerprint) pour éviter que le même utilisateur voie des résultats qui changent aléatoirement, puis je comparerais qualité, coût (tokens) et latence. En parallèle, je mettrais une boucle de feedback utilisateur simple (note 1–5, commentaire, "utile/pas utile"), reliée à analysis_id, afin d'identifier ce qui doit être ajusté dans les filtres, les règles ou les prompts. Enfin, je garderais une trajectoire d'évolution claire : ajout de nouvelles sources via une interface de "tools" stable, normalisation plus robuste (devises, taxes, état neuf/occasion), déduplication de produits (entity resolution) et ajout d'un mode d'explicabilité pour justifier les recommandations.