



je veux réutiliser ograph2 pour créer un RAG hybride.
je veux que cela soit conçu comme des briques réutilisables, avec des formats d'entrée et de sortie de chaque brique qui soient bien définis

Très bien. Voici une proposition d'architecture « RAG hybride » pour réutiliser **ograph2** sous forme de briques clairement séparées, chacune avec des formats d'entrée/sortie explicites. J'ai aligné les interfaces sur ce qui existe déjà dans votre repo (modes **rag**, **kg-rag**, **graphrag**, **ontograph-rag**, etc.) et sur votre design OG-RAG vectoriel→graphe→génération→vérification factuelle. 🤖 🤖

1) Vue d'ensemble des briques

1. Config Loader

Lit un YAML unique décrivant modèle, embeddings, données, méthode de requête et hyperparamètres. Exemple de champs déjà présents : `model`, `embedding_model`, `data`, `query.method`, `query.hyperparams.top_k`, `evaluator`. 🤖 🤖 🤖 🤖

2. Ingestion & Indexation (Vector + OG)

Charge documents, construit/charge l'index vectoriel, prépare les ressources ontologie/OG (noeuds/chunks/embeddings). Vos utilitaires `create_or_load_index`, `get_documents` sont déjà utilisés dans `query_llm.py`. 🤖 🤖 🤖

3. Réécriture de requête (option)

Normalise et enrichit la question utilisateur (débruitage, expansion de mots-clés).

4. Récupération Hybride

- a) top-K vectoriel
- b) expansion guidée par graphe/ontologie (OG) pour construire un sous-graphe/contextualisation. C'est la logique OG-RAG décrite pour « Run 6 ». 🤖
- NB : vous avez déjà un moteur OntoGraph/HyperGraph pouvant servir de base. 🤖

5. Génération

Compose la réponse à partir du contexte (chunks + faits OG) via LLM. Les prompts RAG sont présents dans `OntoGraphQueryEngine`. 🤖

6. Vérification factuelle (LLM-as-Judge)

Valide chaque assertion vs sous-graphe/contextes sélectionnés. Spécifié dans votre plan OG-RAG. 🤖

7. Citations & Traçabilité

Construit des références robustes (URI, chemins OG/Doc). Attendu par votre DoD. 🤖

8. Mémoire (option)

Résume et journalise l'échange (pour relancer la réécriture, la personnalisation, etc.).

9. Évaluation

Génère des jeux Q/R, exécute, calcule des métriques (faithfulness, relevancy, etc.), déjà prévues dans vos configs. 🤖

2) Interfaces standardisées (I/O)

2.1. Schéma de configuration (entrée)

```
yaml



model:
  api_type: "openai"
  api_base: "..."
  api_key: "..."
  deployment_name: "gpt-4o"
  api_version: "2024-02-01"

embedding_model:
  api_type: "openai"
  api_base: "..."
  api_key: "..."
  deployment_name: "text-embedding-3-small"
  api_version: "2024-02-01"

data:
  documents_dir: "data/.../corpus"
  ontology_path: "data/ontology/...jsonld"
  kg_storage_path: "data/kg/.../"
  index_dir: "index/.../"
  subdir: false
  smart_pdf: true
  full_text: true
  chunk_size: 2048 # si vous activez le couplage chunks/OG

query:
  method: "rag" # "kg-rag" | "graphrag" | "ontograph-rag" | "ontohypergraph-rag" | "raptor-
rag"
  mode: "json"
  hyperparams:
    top_k: 5
    graph_expansion:
      depth: 1
      rel_types: ["MENTIONS", "BELONGS_TO"]
      node_labels: ["Fact", "Doc"]
      max_nodes: 200

evaluator:
  reference_free: true
  type: "single"
  metrics: ["faithfulness", "answer_relevancy", "context_precision", "context_recall"]
```

Les champs proposés prolongent vos fichiers **config_*.yaml** existants (mêmes noms/structures).  

2.2. Normalisation des documents (sortie de l'ingestion)



```
json

{
  "doc_id": "uri_ou_hash",
  "title": "titre",
  "text": "contenu intégral",
  "chunks": [
    {
      "chunk_id": "doc_id#c001",
      "text": "texte chunk",
      "meta": {"page": 12, "section": "1.2", "uri": "file:///..."}
    }
  ],
  "meta": {"source": "pdf|web|db", "uri": "file:///..."}
}
```

2.3. Noeuds/rerelations OG (sortie de l'étape OG)

```
json

{
  "nodes": [{ "id": "n1", "label": "Fact", "props": { "key": "...", "value": "..."} }],
  "rels": [{ "id": "r1", "type": "MENTIONS", "start": "n1", "end": "n2", "props": {} }],
  "embeddings": { "n1": [...], "n2": [...] },
  "chunks_map": { "n1": "doc_id#c001" }
}
```

Ce format couvre vos constructions hypergraphe/ontographe pour retrouver chunks pertinents à partir des hyperedges et des hypernodes.  

2.4. Requête standardisée


```
json

{
  "query_id": "uuid",
  "text": "Question utilisateur",
  "user_ctx": { "lang": "fr", "domain": "news|agri|..."},
  "constraints": { "date_from": null, "date_to": null }
}
```

2.5. Réponse du retriever hybride (sortie)

```
json

{
  "query_id": "uuid",
  "vector_hits": [
    { "chunk_id": "doc_id#c001", "score": 0.78 },
    { "chunk_id": "doc_id#c014", "score": 0.72 }
  ],
  "graph_subgraph": {
    "nodes": [...],
    "rels": [...],
    "uris": [ "file:///...", "neo4j://..." ],
    "meta": { "expansion": { "depth": 1, "rel_types": [ "MENTIONS" ], "max_nodes": 200 } }
  },
  "context": {
    "chunks": [ "texte c001", "texte c014" ],
    "facts": [ { "k": "...", "v": "..."}, { "k": "...", "v": "..."} ]
  }
}
```

Le paramètre **expansion** correspond exactement à votre spécification OG-RAG (« Interfaces minimales » de Run 6). 

2.6. Entrée/sortie Génération

Entrée

```
json

{
  "query_id": "uuid",
  "query_text": "Question utilisateur",
  "context": {
    "chunks": [...],
    "facts": [{"k": "...", "v": "..."}]
  },
  "rules": []
}
```

Sortie

```
json

{
  "query_id": "uuid",
  "answer": "texte",
  "citations": [
    {"span": "...", "uri": "file:///...", "evidence": "doc_id#c001"},
    {"span": "...", "uri": "neo4j://node/n1", "evidence": "Fact: ..."}
  ]
}
```

La logique de prompt RAG et la concaténation contexte+rules figurent dans votre moteur OntoGraph.



2.7. Entrée/sortie Vérification factuelle

Entrée

answer + graph_subgraph/context.

Sortie

```
json

{
  "pass": true,
  "issues": [],
  "fixes": ""
}
```


C'est le contrat « LLM-as-a-Judge » proposé dans votre plan. 

2.8. Journalisation/Mémoire

```
json

{
  "query_id": "uuid",
  "question": "...",
  "final_answer": "...",
  "context_refs": ["doc_id#c001", "neo4j://node/n1"],
  "metrics": {"latency_ms": 1234, "tokens_in": ..., "tokens_out": ...}
}
```

2.9. Évaluation (entrée/sortie)

Vos QnA_IO lisent/écrivent CSV/JSON de tests ; conservez ce format, avec colonnes **question**, **answer**, **context**, **score_***. 

Les métriques de foi/relevancy/etc. sont déjà paramétrées dans **evaluator**. 

3) Orchestration de bout en bout

Vous pouvez formaliser un **contrôleur** très simple (pseudo-code), qui réutilise vos classes existantes (sélection de moteur par **query.method**) :

```
python

def answer_question(cfg, user_query):
    # 1) service context, LLM & embeds
    service_context = create_service_context(cfg.model, cfg.embedding_model)
    llm, embeds = load_llm_and_embeds(cfg.model, cfg.embedding_model)


    # 2) documents & index
    docs = get_documents(cfg.data.documents_dir, subdir=cfg.data.subdir,
    smart_pdf=cfg.data.smart_pdf, full_text=cfg.data.full_text)
    vector_index = create_or_load_index(cfg.data.index_dir, service_context, documents=docs)

    # 3) retriever hybride
    # 3a) vector top-K (cfg.query.hyperparams.top_k)
    # 3b) expansion graphe (cfg.query.hyperparams.graph_expansion)
    retrieval = hybrid_retrieve(vector_index, cfg.data.kg_storage_path, embeds,
    cfg.query.hyperparams)

    # 4) génération
    gen = generate_with_context(llm, user_query.text, retrieval.context, rules=[])

    # 5) vérification factuelle
    verdict = judge_answer(gen.answer, retrieval.graph_subgraph)

    # 6) post-traitement citations + mémoire
    return finalize(gen, verdict, retrieval)
```

Les points 1–2–sélection moteur sont déjà dans **query_llm.py** (selon **query.method**). 

4) Contrats de fonctions minimaux (à caler dans src/reason/)


Conformes à votre « Run 6 » :

```
python

def hybrid_search(session, query_vec: list[float], k: int, expansion: dict) -> dict:
    """Retourne un sous-graphe {nodes, rels, uris, meta} + contexte textuel."""

python

def judge_answer(answer: str, context_subgraph: dict) -> dict:
    """{pass: bool, issues: [str], fixes: str}"""
```

Ces signatures sont exactement celles précisées dans votre doc. 

5) Arborescence recommandée

```
perl

configs/
  rag/
    config_<domaine>.yaml
src/
  ingest/
  index/
  og/                                # ontologie/OG utils
  reason/
    c2_retrieval_hybrid.py
    c4_fact_check.py
  gen/
  eval/
scripts/
  ask_v2.py                          # chaîne complète (v2)
  build_kb_v2.py
tests/
  e2e/
```

ask_v2.py orchestrera la chaîne réécriture → vectorisation → **hybrid_search** → textualisation → génération → **judge_answer** → citations → mémoire, comme prévu. 🔄

6) Paramètres clés et bonnes pratiques

- **top_k vectoriel** et **contrôle d'expansion** (profondeur, types de relations, cardinalités) sont les deux leviers-qualité majeurs pour la précision/couverture. Gardez-les visibles dans `query.hyperparams`. 🔄
- **Traçabilité** : stocker systématiquement **uri** et **chunk_id** dans les citations. Votre DoD l'exige. 🔄
- **Évaluation continue** : activez les métriques **faithfulness**, **context_precision/recall**, déjà prévues. 🔄
- **Réutilisabilité** : faites transiter toutes les briques par des **DTO** JSON/Pydantic conformes aux schémas ci-dessus, pour pouvoir plugger/déplugger les moteurs (**rag**, **graphrag**, **ontograph-rag**) au niveau de la brique « retrieval hybride ». Les moteurs existants d'ograg2 seront appelés derrière une interface unique. 🔄 🔄

Souhaitez-vous que je vous produise :

1. un **exemple complet** de `config_hybride.yaml` avec `graph_expansion`,
2. un **squelette** de `c2_retrieval_hybrid.py` et `c4_fact_check.py` prêt à intégrer vos moteurs existants,
3. un `ask_v2.py` minimal qui enchaîne les briques ?



Sources



Si je comprends bien, tu proposes de reprendre quelques sources telquels ?

Exactement. Mon idée n'est pas de tout réinventer, mais de **capitaliser sur les briques déjà présentes dans ograg2** et de les encapsuler pour former une chaîne réutilisable.

Concrètement :

- **À reprendre tel quel (ou presque)**
 - Les loaders (`get_documents`, `create_or_load_index`) qui gèrent ingestion et indexation.
 - La logique de sélection du moteur (`rag`, `kg-rag`, `ontograph-rag`, etc.) déjà dans `query_llm.py`.
 - Les prompts et moteurs spécialisés (par ex. `OntoGraphQueryEngine`) qui savent orchestrer la génération à partir du graphe.
 - La configuration YAML telle que vous l'avez conçue (`model`, `embedding_model`, `data`, `query`, `evaluator`).
- **À encapsuler/refactoriser**
 - La récupération hybride : au lieu d'avoir `rag` ou `kg-rag` appelés séparément, créer une fonction `hybrid_retrieve(...)` qui combine vectoriel + expansion graphe. Cette fonction peut appeler vos moteurs existants en interne.
 - La vérification factuelle (`judge_answer`) : aujourd'hui décrite dans vos docs, à implémenter comme brique réutilisable.
 - Les formats d'entrée/sortie : imposer des DTO (JSON/Pydantic) clairs, pour que chaque brique prenne en entrée/sortie le même type de structure, même si derrière on réutilise vos classes OG déjà codées.

En résumé :

- Oui, certaines **sources sont réutilisées telles quelles** (indexation, moteurs existants).
- Mais vous les **encapsulez derrière des interfaces standardisées** (I/O définis) pour pouvoir brancher/débrancher les moteurs sans toucher au reste de la pipeline.

Souhaitez-vous que je vous montre directement un **exemple de squelette de `c2_retrieval_hybrid.py`** qui illustre comment réutiliser vos moteurs actuels tout en respectant le format d'entrée/sortie que j'ai décrit ?



explique rag, kg-rag, et ontograph-rag

Voici une clarification des trois variantes que vous avez dans **ograg2** :

1. RAG (Retrieval-Augmented Generation)

- **Principe** : le modèle interroge une base vectorielle (ex. Chroma) qui stocke les embeddings des documents.
- **Pipeline** :
 1. Embedding de la question,
 2. Recherche des chunks les plus proches,
 3. Envoi de ces contextes au LLM pour générer la réponse.

- **Cas d'usage** : lorsqu'on a des documents textuels bien structurés, et qu'on veut retrouver rapidement les passages pertinents.
 - **Limite** : ne gère pas les relations complexes entre entités (se limite à la proximité sémantique).
-

2. KG-RAG (Knowledge Graph RAG)

- **Principe** : au lieu d'uniquement interroger les embeddings, on exploite un **graphe de connaissances** construit à partir du corpus.
 - **Pipeline** :
 1. Extraction des entités et relations lors de l'ingestion (NER + parsing des relations),
 2. Stockage dans Neo4j (ou équivalent),
 3. Lors de la requête : expansion des nœuds pertinents (chemins, voisins),
 4. Envoi des contextes enrichis (relations, attributs) au LLM.
 - **Cas d'usage** : quand les liens entre concepts sont importants (ex. relations hiérarchiques, dépendances).
 - **Avantage** : permet des réponses explicatives et structurées.
 - **Limite** : plus coûteux à maintenir (construction et mise à jour du graphe).
-

3. OntoGraph-RAG

- **Principe** : une spécialisation du KG-RAG où le graphe n'est pas « brut » mais aligné sur une **ontologie** définie (ex. une taxonomie métier).
 - **Pipeline** :
 1. Construction du graphe avec des concepts reliés à une ontologie métier,
 2. Requêtes orientées par cette ontologie (filtres sémantiques),
 3. Utilisation de moteurs spécialisés (**OntoGraphQueryEngine**) qui traduisent la requête utilisateur en chemin logique sur le graphe,
 4. Passage au LLM pour formuler la réponse.
 - **Cas d'usage** : quand il faut raisonner avec des **catégories normées** (par ex. compétences, processus, standards).
 - **Avantage** : garantit de la cohérence avec le vocabulaire métier.
 - **Limite** : rigidité si l'ontologie est trop stricte.
-

Synthèse :

- **RAG** = rapide et simple, basé sur la similarité vectorielle.
- **KG-RAG** = ajoute la structure des relations (réseau de concepts).
- **OntoGraph-RAG** = ajoute une couche d'ontologie, pour raisonner dans un cadre normatif métier.

Voulez-vous que je vous propose un **schéma en briques réutilisables** (I/O bien définis) montrant comment un pipeline hybride peut combiner les trois approches selon les cas d'usage ?



Sources



si je veux un avantage concurrentiel mais aussi avoir de la flexibilité pour implémenter, est il réaliste de viser OntoGraph-RAG ?

C'est réaliste, mais avec des nuances importantes à bien mesurer :

Avantages concurrentiels d'OntoGraph-RAG

- **Différenciation forte** : très peu d'acteurs vont jusqu'à intégrer une ontologie dans un RAG. Cela vous positionne comme apportant non seulement de la recherche, mais aussi une **couche de raisonnement métier**.
- **Confiance et explicabilité** : les utilisateurs comprennent mieux la réponse si elle est ancrée dans des concepts normalisés (« cette compétence relève du domaine X selon l'ontologie »).
- **Barrière à l'entrée** : une fois le graphe et l'ontologie construits, vos concurrents auront plus de mal à reproduire ce socle sans effort similaire.

Contraintes et réalités de mise en œuvre

- **Complexité initiale** : il faut soit créer, soit réutiliser une ontologie métier solide. Sans cela, le système sera vite limité.
- **Coût de maintenance** : les ontologies évoluent avec les métiers ; il faut donc prévoir des mécanismes de mise à jour semi-automatique (par ex. extraction de nouveaux concepts et alignement).
- **Rigidité** : si l'ontologie est trop stricte, le système devient moins souple que le RAG vectoriel pur.

Stratégie pragmatique

1. **Commencer par un RAG hybride (vectoriel + KG-RAG)** : vous gagnez déjà en richesse et en pertinence des réponses.
2. **Ajouter une surcouche ontologique ciblée** : au lieu d'ontologiser tout le graphe, concentrez-vous sur quelques domaines critiques (par ex. compétences, processus clés).
3. **Évoluer progressivement** vers un OntoGraph-RAG complet, en gardant la flexibilité d'interroger soit le vectoriel, soit le graphe, soit l'ontologie selon le cas d'usage.

En résumé : viser OntoGraph-RAG est réaliste et différenciant, mais la bonne approche est **progressive**. Le plus compétitif est de **capitaliser sur une base hybride** et d'**ontologiser par couches**, ce qui vous laisse la flexibilité d'implémentation sans bloquer le projet dans la lourdeur d'un graphe entièrement normé dès le départ.

Souhaitez-vous que je vous propose un **plan de montée en maturité** (par étapes concrètes) pour passer de RAG → KG-RAG → OntoGraph-RAG ?



Quels sont les types d'architectures de rags les plus fréquemment implémentés aujourd'hui ?

Aujourd'hui, dans les implémentations industrielles, on retrouve surtout quelques grandes familles d'architectures de RAG, avec des variantes plus ou moins avancées :

1. RAG vectoriel classique (le plus courant)

- **Principe** : stockage des documents en embeddings dans une base vectorielle (Chroma, Pinecone, Weaviate, FAISS).
- **Pipeline** : encodage → recherche par similarité → passage des chunks au LLM.
- **Pourquoi répandu** : facile à mettre en place, maintenable, peu coûteux, bonne performance dans des cas génériques.
- **Limite** : perte de précision sur des requêtes complexes ou relationnelles.

2. RAG avec reranking / multi-step retrieval

- **Principe** : on enrichit le retrieval vectoriel par un reranker (souvent un modèle cross-encoder type Cohere Rerank ou bge-reranker) qui réordonne les résultats.
- **Avantage** : meilleure pertinence contextuelle.
- **Usage fréquent** : bases documentaires avec beaucoup de bruit, FAQs, knowledge bases internes.

3. RAG hybride (vectoriel + lexical ou BM25)

- **Principe** : combiner recherche dense (embeddings) et recherche lexicale (BM25/Elasticsearch).
- **Avantage** : équilibre entre recherche sémantique et recherche exacte (mots-clés, chiffres, dates).
- **Très courant** : adopté dans beaucoup d'architectures d'entreprise, car il limite les trous de rappel du vectoriel pur.

4. RAG multi-granularité

- **Principe** : indexation à plusieurs niveaux (chunks, paragraphes, documents, métadonnées).
- **Pipeline** : on commence large (retrieval de documents), puis on raffine (retrieval intra-document).
- **Usage** : documentation technique, conformité réglementaire.

5. RAG structuré (ou "Graph-RAG")

- **Principe** : construction d'un graphe de connaissances (Neo4j, ArangoDB) à partir du corpus, avec entités et relations.
- **Avantage** : permet des requêtes complexes, explicatives, et des liens multi-documents.
- **Adoption** : encore minoritaire, mais en forte croissance dans les secteurs à forte complexité (santé, finance, juridique).

6. RAG avec agents / planification

- **Principe** : le LLM agit comme un orchestrateur, choisissant dynamiquement les sources à interroger (vector DB, SQL, API, graphe, etc.).
- **Avantage** : plus flexible et modulaire.
- **Exemple** : LangChain Agents, LlamaIndex RouterQueryEngine.
- **Tendance** : de plus en plus exploré pour des use cases complexes.

7. RAG enrichi par génération intermédiaire (Chain-of-Thought RAG)

- **Principe** : le LLM génère d'abord une requête reformulée (ou plusieurs variantes), puis celles-ci alimentent la recherche.
- **Avantage** : améliore le rappel et la robustesse.
- **Usage** : systèmes où les utilisateurs posent des questions vagues, incomplètes, ou multilingues.

En résumé :

- **Le plus fréquent aujourd'hui en production** : RAG vectoriel simple, RAG hybride (vectoriel + lexical), et RAG avec reranking.
- **En forte progression** : Graph-RAG et RAG orchestrés par agents.
- **Encore niche** : OntoGraph-RAG, mais très différenciant dans des contextes spécialisés.

Voulez-vous que je vous dresse un **tableau comparatif synthétique** (type "forces / limites / maturité marché") pour choisir lequel viser selon vos priorités ?



qu'est ce qui est le plus réaliste, en termes de d'infrastructures et de ressources matérielles, si mes clients veulent être hébergés sur des serveurs locaux, du fait de besoin de souveraineté des données imposée par la situation insulaire de la Nouvelle-Calédonie?

Voici une réponse pragmatique, pensée pour un hébergement **on-prem** en Nouvelle-Calédonie, avec des contraintes de souveraineté, de connectivité et de maintenance locale.

1) Architecture cible réaliste (priorité au simple & robuste)

A. "RAG hybride" standard (recommandé pour 80 % des cas)

- **Fonctions** : BM25 + recherche vectorielle + reranking.
- **Piliers logiciels (100 % on-prem)** :
 - **Index lexical** : OpenSearch/Elasticsearch (BM25) ou Meilisearch.
 - **Vector DB** : Qdrant (faible empreinte, binaire unique) ou pgvector (PostgreSQL).
 - **LLM d'inférence** :
 - GPU : vLLM ou TGI (text-generation-inference).
 - CPU ou GPU léger : llama.cpp (quantisés GGUF).
 - **Reranker** : bge-reranker-base (ou équivalent local).
 - **API/orchestration** : FastAPI + Nginx (reverse proxy), Docker Compose.
 - **Ingestion** : pdfplumber, unstructured (option), Tesseract OCR si scannés.
- **Avantages** : mise en place rapide, maintenance simple, très bonnes performances pour QA documentaire.
- **Évolutif** : on peut ajouter plus tard un graphe de connaissances (KG) ou une couche ontologique.

B. "Graph-RAG" ciblé (pour domaines très relationnels)

- **Ajout** : Neo4j/ArangoDB pour entités-relations, pipeline d'extraction (NER + relations).
- **Usage** : conformité, référentiels, dépendances multi-documents.
- **Coût** : plus de MCO (maintenance en condition opérationnelle). À limiter aux périmètres critiques.

C. "OntoGraph-RAG" sélectif (avantage concurrentiel, périmètre borné)

- **Ajout** : gestion d'une ontologie métier (JSON-LD/OWL), alignement périodique.
- **Stratégie** : ontologiser 1-2 domaines à fort impact (ex. compétences/processus), pas tout le SI.

2) Dimensionnement matériel minimal par palier

Palier	Cas d'usage	Serveur conseillé	LLM	Débit typique (indicatif)
S-0 (CPU-only)	POC, faible charge, < 10 users	16-24 vCPU, 64-128 Go RAM, SSD NVMe	7-8B quantisé (GGUF) via llama.cpp	5-15 tok/s
S-1 (GPU "atelier")	Équipe interne, < 30 users	1× GPU 24 Go (RTX 4090/A5000), CPU 16 vCPU, 128 Go RAM	7-8B (FP16/INT4), vLLM/TGI	70-150 tok/s
S-2 (prod locale)	> 50 users, multi-projets	2× GPU 24-48 Go ou 1× L40S 48 Go, 32 vCPU, 256 Go RAM	8-13B (voire Mixtral 8×7B INT4 partiel)	150-300 tok/s
S-3 (haute charge)	Centre de service	2-4× GPU 48-80 Go, 64 vCPU, 512 Go RAM	13B-70B (INT4/FP8, selon VRAM)	> 300 tok/s

Remarques pratiques :

- **7-8B** couvre déjà l'essentiel des besoins QA internes, surtout avec **reranking** et **prompts rigoureux**.
- **INT4** permet 7-8B sur 16-24 Go VRAM ; **FP16** nécessite ~14-16 Go VRAM pour 7-8B.
- **vLLM** offre un très bon **throughput** pour servir plusieurs utilisateurs concurrents.

3) Stockage et volumétrie (ordre de grandeur)

- **Embeddings** (Float32) :
 - 100 000 chunks × 768 dims × 4 o ≈ **293 Mo** (sans métadonnées).
 - 1 000 000 chunks ≈ **2,9 Go** (compter 5-8 Go avec métadonnées).
 - Avec **quantization/PQ** (Qdrant/Milvus), on peut diviser par 2-4 selon la méthode.
- **Documents** : prévoir **1-3×** la taille brute (PDF + textes + caches + OCR).
- **Index lexical** : taille proche du brut textuel compressé (varie selon moteur et analyzers).
- **Sauvegardes** : snapshot quotidien (vector DB + index lexical), **restic** vers stockage chiffré.

4) Réseau, sécurité, souveraineté

- **Totalement local** : pas de dépendance cloud ; mises à jour “déconnectées” possibles (artifacts Docker/DEB/RPM).
- **Chiffrement au repos** : LUKS/dm-crypt, chiffrement natif DB si dispo.
- **Chiffrement en transit** : TLS interne (Nginx ↔ services).
- **Cloisonnement** : VLAN/pare-feu, bastion SSH.
- **Identités** : SSO local (Keycloak/LDAP), RBAC par projet.
- **Coffre-fort** : HashiCorp Vault pour secrets/API keys internes.
- **Journalisation** : audit des requêtes (PII), conservation locale (ELK/Opensearch + WORM si nécessaire).

5) Pile logicielle recommandée (tout local)

- **Recherche** : OpenSearch (BM25) + Qdrant (vectors) + reranker.
- **Serveur d'IA** : vLLM (GPU) ou llama.cpp (CPU/GPU mixte).
- **Orchestration RAG** : FastAPI (back-end), workers (Celery/RQ) pour ingestion & OCR.
- **ETL/ingestion** : pipelines Dockerisés ; OCR Tesseract si besoin ; “smart-PDF” pour segmentation.
- **Supervision** : Prometheus + Grafana ; alertes simples (mail/on-prem SMTP).
- **Déploiement** : Docker Compose au début ; Ansible pour reproductibilité ; possible passage à K3s si nécessaire.

6) Choix de modèles locaux (sans cloud)

- **LLM** : Llama 3.1-8B Instruct, Mistral-7B-Instruct, Qwen2-7B-Instruct (quantisés si besoin).
- **Embeddings** : bge-m3 / nomic-embed-text / e5-base (selon langues).
- **Reranker** : bge-reranker-base (cross-encoder).
- **Langues** : si besoin FR/EN, privilégier bge-m3 (multi-domaines, multilingue).

7) Haute disponibilité minimale (facile à maintenir)

- **Vector DB** : Qdrant en **3 nœuds** (replicas = 2) sur mêmes racks ou baies séparées.
- **OpenSearch** : 3 nœuds master-eligible + 1 data (selon volumétrie).
- **LLM** : 1 nœud actif + 1 passif (image miroir, warm-standby).
- **Sauvegardes** : snapshots quotidiens hors bande (NAS local, bande si disponible).

8) Coûts et réalités terrain

- **Énergie & refroidissement** : privilégier GPUs à bon perf/W (L40S) si budget ; sinon RTX 4090 (civile) pour atelier.
- **Disponibilité pièces** : garder **1 GPU de spare** si production critique.
- **Simplicité** : la valeur vient surtout d'un **RAG hybride bien réglé** (BM25 + vectors + rerank) avec une bonne ingestion et un prompt maîtrisé. Le KG/ontologie doit rester **modulaire** et limité à des périmètres à ROI clair.

9) Feuille de route concrète

1. V1 (4–6 semaines) :

- Stack **BM25 + Qdrant + reranker** ; LLM 7–8B local ; ingestion PDF/OCR ; métriques de base.
- SLA interne simple (heures ouvrées), sauvegardes, monitoring.

2. V2 :

- Ajout de **patterns multi-granularité** (document→paragraphe→phrase), filtres par métadonnées, améliorations UI.
- Tests de charge, optimisation des prompts.

3. V3 :

- **KG** ciblé sur un domaine (ex. compétences/processus) + quelques “cartouches” d'OntoGraph sur le même domaine.
- Évaluation formelle (faithfulness, recall, précision contextuelle), ajustements.

Conclusion

Le plus réaliste, en on-prem insulaire, est de déployer d'abord un **RAG hybride** (BM25 + vecteurs + rerank) servi par un **LLM 7–8B local** (GPU unique 24 Go ou CPU-only quantisé si contrainte budgétaire).

Cette base est **pérenne, souveraine, maintenable** et couvre la majorité des besoins. Ensuite, **ajoutez un KG/ontologie de façon ciblée** là où la valeur métier justifie la complexité supplémentaire.



Pour les RAG de juristes d'entreprise, dans des grandes sociétés industrielles ayant des clients B2B mais aussi B2C (fournisseurs d'électricité par exemple), quelles seraient les architectures les plus réalistes et efficaces, toujours dans le même contexte

Voici ce qui est réaliste et efficace pour un **RAG juridique on-prem** dans une grande entreprise industrielle (B2B + B2C, ex. électricité), avec exigences fortes de souveraineté des données et de disponibilité locale.

1) Architecture de base recommandée (80 % des besoins)

RAG hybride : recherche lexicale (BM25) + recherche dense (embeddings) + reranking cross-encoder.

- **Ingestion & préparation**

- Connecteurs DMS/NAS (PDF, DOCX, emails exportés), OCR si nécessaire.
- Normalisation, découpe multi-granularité (document → section → paragraphe → phrase).
- Classification documentaire (contrat, avenant, courrier AR, décision régul., guide interne).
- Extraction de métadonnées juridiques utiles à la recherche : type, parties, juridiction, champ géographique, **date(s) d'effet et de publication**, clauses (SLA/force majeure/pénalités), références normatives.
- Détection/masquage PII (NIR, adresse, IBAN...), journalisation des traitements.

- **Indexation**

- **Lexical** : OpenSearch/Elasticsearch (analyzers FR/EN), filtres stricts (juridiction, période d'effet, type de doc).
- **Vectorel** : Qdrant ou pgvector (PostgreSQL) ; embeddings multilingues (FR/EN).
- **Reranker** : cross-encoder (type bge-reranker) pour réordonner le top-K issu de BM25+dense.

- **Retrieval**

- Stratégie **hybride**: (BM25 u Dense) → fusion scorée → **rerank**.
- Réécriture de requête (synonymes juridiques, expansions contrôlées).
- **Filtres obligatoires** avant dense (ex. « droit de la consommation », « décret X », « applicable au 01/01/2023 »).

- **Génération/Restitution**

- LLM local 7-8B (FR/EN) avec gabarits de prompts juridiques.
- **Sortie structurée**: réponse, **citations ancrées** (doc, page, §), passages sources, **niveau de confiance**, motifs d'incertitude.
- Mode « **extraction stricte** » (quotes + reformulation minimale) pour réduire le risque d'hallucination sur sujets sensibles.

- **Vérification & gouvernance**

- Vérification factuelle guidée par les citations (alignement passage ↔ affirmation).
- Journal d'audit : requêtes, sources consultées, versions de documents.
- Cloisonnement par **espace de noms** (BU/filiale) et **RBAC**.

- **Pourquoi cette base ?**

- Performante et **maintenable on-prem**.
- Très bonne précision sur Q/R contractuelles et réglementaires grâce aux **filtres métier + reranking**.
- Évolutive vers le graphe/ontologie sans refondre la pile.

2) Extensions ciblées pour juristes d'entreprise

A. Versioning temporel « droit applicable »

- Index séparé « normes » (lois, décrets, décisions régul., CGV standard) avec **plages de validité** (`effective_from`, `effective_to`), juridiction et références.
- Le moteur force un **filtrage par date d'effet** + juridiction avant toute similarité sémantique.
- Cas d'usage : « Quel texte était applicable à la date de signature du contrat X ? »

B. Bibliothèque de clauses & “playbooks” de négociation

- Extraction/normalisation de **clauses types** (SLA, pénalités, résiliation, RGPD, force majeure).
- Index vectoriel des clauses + règles d'évaluation (« écart vs clause standard », seuils d'acceptation).
- Cas d'usage : **revue d'un contrat** B2B, score d'écart, suggestions de reformulation.

C. Petit Graph-RAG focalisé (obligations, parties, montants, délais)

- Graphe léger pour relier **Partie** → **Obligation** → **Événement** → **Pénalité** → **Échéance**.
- Requêtes explicatives : « Quelles pénalités s'appliquent si l'indisponibilité > 4 h sur contrat Y ? »
- Déploiement **progressif** : seulement sur les contrats critiques (réseaux, fourniture, MCO).

D. Ontologie restreinte (facultatif, différenciant)

- Vocabulaire contrôlé pour 3–5 **catégories métier** (Obligation, Clause, Pénalité, DonnéePersonnelle, Norme).
- Sert de couche de **validation** (tags cohérents) et d'**explication** aux utilisateurs.
- À introduire après stabilisation de la base hybride.

3) Pile logicielle on-prem réaliste

- **Recherche** : OpenSearch/Elasticsearch (BM25) + Qdrant/pgvector.
- **LLM serveur** : vLLM (GPU) ou llama.cpp (CPU/GPU) selon budget/perf.
- **Embeddings** : multilingues (FR/EN) type bge-m3 / e5-base-fr-en.
- **Reranker** : bge-reranker-base.
- **Pipelines** : FastAPI pour l'orchestration, Celery/RQ pour l'ingestion, Tesseract OCR si scannés.
- **Sécurité** : Keycloak/LDAP, TLS interne, chiffrement au repos, coffre-fort de secrets.
- **Supervision** : Prometheus + Grafana ; snapshots réguliers des index.

4) Dimensionnement matériel indicatif (par palier)

- **S-1 (atelier/équipe)** : 1× GPU 24 Go (ou CPU only quantisé), 16–32 vCPU, 128 Go RAM.
- **S-2 (production locale)** : 2× GPU 24–48 Go ou 1× 48 Go, 32 vCPU, 256 Go RAM.
- Vise un LLM **7–8B** pour servir les usages QA/contrats en FR/EN avec **reranking**.

5) Qualité, conformité et MCO

- **Évaluation continue** : recall@k, précision contextuelle, **faithfulness**, temps de réponse, taux de citations valides.
- **Modes sûrs** : “no-gen” (retrieval-only) pour recherches sensibles ; “quote-first” pour avis rapides.
- **Traçabilité** : conserver empreintes des documents et **versions** (hash + date).
- **PII & secret d'affaires** : masquage sélectif, périmètres d'accès, rétention locale.

6) Feuille de route pragmatique

1. **V1** – Hybride robuste : BM25 + vecteurs + reranker, filtres juridiction/période/type, citations page/paragraphe, PII.
2. **V2** – Spécialisation juridique : versioning des normes, bibliothèque de clauses, gabarits de réponses, “quote-first”.
3. **V3** – Ciblage avancé : petit **Graph-RAG** sur obligations et pénalités des contrats critiques ; ontologie restreinte si besoin d'explicabilité accrue.

En synthèse

Pour des juristes d'entreprise en contexte B2B/B2C et souveraineté stricte, **le meilleur rapport valeur/effort** est un **RAG hybride** enrichi de filtres métier, de **versioning des normes** et d'une **bibliothèque de clauses**. On ajoute ensuite, là où le ROI est clair, un **Graph-RAG ciblé** sur les obligations/pénalités. L'ontologie n'est pas nécessaire d'emblée, mais devient un atout d'explicabilité une fois la base stabilisée.



OGRAG2 est un référentiel de qualité, bien développé, créé par une entreprise renommée et sous licence ouverte. En matière de repository de référence, Quelles options s'offrent à moi si je m'oriente vers un RAG hybride (recherche lexicale (BM25), recherche dense (embeddings) et reranking cross-encoder), tout en gardant à l'esprit qu'il faut que ce soit évolutif ?

Voici des **options de “repositories de référence”** et de **pires open source** solides pour un **RAG hybride** (BM25 + dense + reranking cross-encoder) qui restent **évolutives** et **on-prem**.

1) Pistes « prêtes à l'emploi » (frameworks RAG)

1. Haystack 2.x (deepset) – pipeline hybride + reranker

- Tutoriaux officiels pour construire un pipeline hybride: BM25 + dense + **ranker cross-encoder** (bge-reranker / SentenceTransformers), avec nœuds de fusion et visualisation du DAG. [Haystack+2](#)
- Avantage: composables, neutre vis-à-vis du stockage (Qdrant, FAISS, OpenSearch, etc.), et recettes prêtes pour RRF/merge + reranking. [Haystack](#)

2. LangChain – EnsembleRetriever (RRF) + Cross-Encoder Reranker

- “EnsembleRetriever” combine BM25 et dense via **Reciprocal Rank Fusion**; vous pouvez ajouter un reranker cross-encoder derrière. [LangChain](#)
- Avantage: très modulable si vous avez déjà des briques LangChain.

2) Pistes « moteur de recherche / DB » (socle durable)

1. OpenSearch (BM25 natif + neural / k-NN + pipelines hybrides)

- Hybride **intégré**: opérateur **hybrid** + **pipelines de normalisation** (min-max, L2) et **RRF**; pondération lexical/dense au runtime. [OpenSearch+1](#)
- Bonnes pratiques et résultats: normalisation + moyenne pondérée, k raisonnable (100–200) pour pertinence/latence. [OpenSearch](#)
- Intérêt: si vous avez déjà un SI orienté “moteur de recherche” (logs, filtres, agrégations), c’est un **socle enterprise-grade**.

2. Qdrant (vector DB légère) + **sparse/BM25 intégrés** + reranking

- Collection hybride: denses + **sparse BM25** au **sein de la même collection**, avec “prefetch” multi-requêtes et fusion côté serveur; support du **reranking** après hybrid retrieval. [Qdrant](#)
- Intérêt: déploiement simple (binaire/Docker), bonne perf on-prem, API claire pour hybrider et réordonner.

3. Weaviate (vector DB) avec **hybrid** + **reranker intégré**

- Hybride: exécution parallèle BM25 + vector, **fusion normalisée** (relativeScoreFusion par défaut) ou **rankedFusion**. [Weaviate DocumentationWeaviate](#)
- Reranking: modules **Transformers/Cohere/VoyageAI**, y compris **hébergés localement** (conteneur Hugging Face pour le reranker). [WeaviateWeaviate Documentation](#)

- Intérêt: “tout-en-un” moderne, simple à faire évoluer (plugins).

4. Milvus 2.5 (vector DB) + Full-Text/BM25 natif

- Introduit **full-text** (sparse) directement dans Milvus → facilite un **hybride** dans un seul socle (“doc in, doc out”). [Milvus Blog](#)

3) Modèles de reranking (cross-encoders) et entraînement

- Vous pouvez démarrer avec des **cross-encoders publics** (p.ex. **BAAI/bge-reranker-base**) et, si besoin, **affiner sur vos données** avec Sentence-Transformers 4.x (tutoriel HF sur le **finetuning de rerankers**). [HaystackHugging Face](#)

4) Trois “blueprints” réutilisables (référentiels types)

A) Blueprint OpenSearch-centré

- **Index lexical** BM25 + **index neural** (embeddings) dans OpenSearch.
- **Search pipeline**: normalisation **min-max** + combinaison pondérée, ou **RRF**; filtres métier par *must* et *post_filter*. [OpenSearch+1](#)
- **Reranking**: service externe (FastAPI) qui applique un cross-encoder sur le top-K.
- Usage: environnements qui veulent **un seul cluster** moteur (recherche + métriques).

B) Blueprint Qdrant + Haystack

- **Qdrant** pour dense + **sparse BM25** dans la **même collection**, requête **prefetch**; **Haystack** pour orchestrer **fusion + reranking**. [QdrantHaystack](#)
- Avantages: simplicité de déploiement, scalabilité horizontale Qdrant, DAG Haystack.

C) Blueprint Weaviate “tout-en-un”

- **Hybrid** (BM25 + vector) avec **relativeScoreFusion**, **reranker Transformers** en conteneur local. [Weaviate DocumentationWeaviate](#)
- Avantages: intégration native du reranking dans la requête, moins de “colle” applicative.

5) Critères de choix (souveraineté, évolutivité)

- **Opérations & MCO**
 - Déjà Elasticsearch/OpenSearch en interne → **OpenSearch hybride**: moindre friction, logs/monitoring déjà en place. [OpenSearch](#)
 - Besoin d'un **vecteur-first simple** → **Qdrant** (léger, efficace) ou **Weaviate** (modules). [Qdrant Weaviate Documentation](#)
- **Performance & latence**
 - Hybride ajoute **6–8 %** de latence vs mono-mode dans les tests OpenSearch (acceptable en général). [OpenSearch](#)
 - Reranking cross-encoder est coûteux: appliquer sur **top-K limité** (20–50) et batcher.

- **Évolution vers KG / Onto**

- Tous ces socles laissent la place à un **layer graphe** ultérieur (Neo4j/Arango), car la couche retrieval reste **contractualisée** (I/O: query → candidats → rerank → contexte).

6) Recommandation pragmatique

- **Si vous voulez une base “industry-grade” et pérenne:**
OpenSearch hybride (BM25 + neural) + **pipeline de normalisation/RRF et reranking cross-encoder** via micro-service. Ajoutez Haystack ou un orchestrateur maison pour le workflow RAG. [OpenSearch+1](#)
- **Si vous voulez la simplicité et un déploiement rapide on-prem:**
Qdrant (hybride + reranking) **piloté par Haystack 2.x**: peu de dépendances, flexible, facile à faire évoluer. [QdrantHaystack](#)
- **Si vous voulez un “tout-en-un” moderne:**
Weaviate (hybride + reranker local intégré) pour réduire la “colle” applicative et garder la modularité. [Weaviate DocumentationWeaviate](#)

Souhaitez-vous que je vous propose un **squelette de repo** (Docker Compose + exemples de requêtes) pour chacun de ces trois blueprints, afin que vous puissiez les évaluer rapidement en environnement souverain ?



je parlais de produits aboutis comme orgarg, qui est très intégré

Merci de la précision. Si vous cherchez des **produits/repositories “aboutis” et intégrés** (au sens de stacks cohérentes, pas seulement des librairies), qui couvrent un **RAG hybride** (BM25 + dense + reranking) et qui restent **évolutifs** et **auto-hébergeables**, voici les options les plus solides aujourd’hui.

1) Pivots “moteur de recherche intégré”

Ces solutions offrent dans le même socle la recherche lexicale, la vectorielle et des mécanismes natifs de fusion/reranking.

- **OpenSearch (2.11+)** – “Hybrid search” via pipelines de recherche, normalisation des scores (min-max, L2) et **RRF** (rank-fusion) ; conception prévue pour mixer clauses BM25 et vector en une requête unifiée. Évolutif (cluster), on-prem simple. [OpenSearch DocumentationOpenSearch](#)
- **Elasticsearch (8.14+ retrievers, 8.16 GA)** – Hybride dense/sparse/lexical, usage de **RRF** via “retrievers”, support ELSER (sparse) et kNN (dense). Mature, nombreuses features d’exploitation. [Elastic+1](#)
- **Vespa** – Stack “search + ranking” très intégrée : BM25 + vector closeness, **Learning-to-Rank**, **ONNX** pour cross-encoders de reranking, et tutoriels hybrides prêts à l’emploi. Excellent pour industrialiser un pipeline de ranking à phases (first/second phase). [docs.vespa.ai+2](#)

Pourquoi les choisir ?

Un seul plan de contrôle pour l'hybride, reranking **dans** le moteur, scaling éprouvé, gouvernance/logs. Ce sont les "références industrielles" si vous voulez une base très intégrée et durable.

2) Pivots "vector DB intégrée hybride"

Solutions orientées base vectorielle qui ont **ajouté** du BM25/sparse et du reranking côté serveur.

- **Qdrant** – Collections **multi-vecteurs** (dense + sparse BM25 + late interaction/ColBERT), **prefetch** multi-branches et **reranking** intégré sur le top-K ; très simple à déployer en on-prem. [Qdrant+2](#)
- **Weaviate** – Hybride (BM25 + vector) avec **relativeScoreFusion/rankedFusion** et **reranker Transformers** hébergé localement (conteneur HF) pour réordonner les résultats. [Weaviate OpenSearch](#)
- **Milvus** (2.5) – Introduit la **full-text/sparse** intégrée, facilitant un pipeline hybride dans le même socle. [OpenSearch Documentation](#)

Pourquoi les choisir ?

Déploiement rapide, API modernes, bonne performance. Idéal si vous partez "vector-first" et souhaitez **ajouter** BM25 + rerank sans tout basculer sur un moteur de recherche complet.

3) Pivots "plateforme RAG" (plus haut niveau)

Stacks "clé en main" pour ingestion → hybride → RAG → citations, avec UI/API.

- **RAGFlow (Apache-2.0)** – Plateforme RAG open-source avec **hybrid recall + fused re-ranking**, ingestion avancée (PDF/Doc layout), citations, Docker on-prem ; utilise Elasticsearch/Infinity en backend document store. [GitHubragflow.io](#)
- **Haystack (OSS / Enterprise / Platform)** – Framework mature avec pipelines hybrides + reranker ; version **Enterprise** et **Platform** existent pour l'exploitation on-prem (support, templates de déploiement). [deepset.ai+1](#)

Pourquoi les choisir ?

Time-to-value rapide (UI, APIs, workflows), tout en permettant de brancher vos moteurs (OpenSearch, Qdrant, etc.).

Recommandations par profils (pour rester "très intégré")

- **Vous avez déjà un historique Elasticsearch/OpenSearch** (logs, SIEM, recherche documentaire) : partez sur **OpenSearch "Hybrid search"** ou **Elasticsearch retrievers + RRF** ; ajoutez un **reranker** cross-encoder (HF/ONNX) et vos filtres métier. C'est le meilleur compromis intégration/évolutivité. [OpenSearch DocumentationOpenSearchElastic](#)
- **Vous voulez un moteur de ranking extensible "search-native"** (L2R, phases, ONNX dans le cœur du moteur) : **Vespa** est la référence. Vous combinez BM25+vector en first-phase, puis **cross-encoder ONNX** en second-phase. Excellent pour des cas complexes et l'A/B test de profils de ranking. [docs.vespa.ai+1](#)

- **Vous privilégiez la simplicité opérationnelle et l'API** (Docker, une DB moderne) : **Qdrant** (hybride + rerank) ou **Weaviate** (hybride + reranker Transformers) ; vous orchestrez la chaîne RAG depuis votre service (FastAPI) ou via **Haystack**. [QdrantWeaviate](#)
- **Vous voulez un "tout-en-un" RAG avec UI** et gouvernance rapide on-prem : **RAGFlow** ou **Haystack Platform/Enterprise** selon votre préférence OSS vs. produit supporté. ragflow.io/deepset.ai

Points d'attention (évolutivité)

- **Reranking** : gardez-le **sur un top-K restreint** (20-50) pour tenir la latence ; tous les socles ci-dessus le supportent via plugin/ONNX/service dédié. [OpenSearch](#)
- **Fusion des scores** : **RRF** est robuste et peu sensible au calibrage ; disponible nativement (OpenSearch/Elastic) ou via pipelines côté DB (Qdrant/Weaviate). [OpenSearch Documentation Elastic](#)
- **Montée en complexité** : ces socles restent compatibles avec un **ajout ultérieur** de graphe/ontologie (Neo4j/Arango) en gardant le contrat I/O "retrieve → rerank → contexte".

En synthèse

Si vous cherchez l'équivalent "très intégré" d'ograg2 côté produit :

- **OpenSearch/Elasticsearch** ou **Vespa** si vous voulez que **tout** (BM25, vector, fusion, rerank) vive **dans le moteur**. [OpenSearch DocumentationElasticdocs.vespa.ai](#)
- **Qdrant/Weaviate** si vous souhaitez une **DB intégrée hybride** simple à exploiter, avec reranking local. [QdrantWeaviate](#)
- **RAGFlow/Haystack Platform/Enterprise** si vous voulez un **produit de niveau plateforme** (UI, API, monitoring) on-prem. ragflow.io/deepset.ai

Si vous me donnez vos contraintes (matériel, volumétrie, SSO, exigences de latence), je peux vous proposer un **squelette de repo** "référence" pour l'option retenue (Docker Compose, exemples de requêtes hybrides, reranking ONNX/cross-encoder, métriques).



si hje développe avec haystack, que permet haystack en quasi clé en mains ?

Voici, de façon pragmatique, ce que **Haystack** vous apporte "quasi clé en main" pour un **RAG hybride** (BM25 + dense + reranking), sans devoir tout réécrire.

Ce que Haystack fournit directement

1. Pipelines composables (DAG)

- Construction d'un **pipeline d'ingestion** (split en chunks, embeddings, écriture en store) et d'un **pipeline de requête** (BM25 + dense → fusion → **reranking cross-encoder**), prêts à instancier et chaîner. [Haystack](#)
- Composants standards : `DocumentSplitter`, `DocumentEmbedder`, `DocumentWriter`, retrievers BM25 & embedding, `DocumentJoiner` (concaténation / RRF), `TransformersRanker` (ex. **bge-reranker-base**). [Haystack+1](#)

2. Hybride “out of the box”

- Tutoriels officiels pas à pas pour **BM25 + dense + reranking** et visualisation du pipeline. [Haystack+1](#)
- Méthodes de **fusion** des résultats (ex. `reciprocal_rank_fusion` via `DocumentJoiner`) ou simple concat + reranking. [Haystack](#)

3. Connectivité multi-backends (Document Stores)

- Support natif ou extensions pour **Elasticsearch/OpenSearch, FAISS, Weaviate, Qdrant, Milvus, Pinecone, SQL**, avec recommandations d'usage selon vos contraintes. [Haystack Documentation+1](#)

4. Évaluation intégrée des RAG

- **Évaluateurs prêts à l'emploi** : *Context Relevance*, *Faithfulness* (LLM-as-judge), *SAS* (similarité sémantique), ainsi que des métriques classiques (Recall, MRR, MAP, EM/F1). Pipelines d'évaluation séparés ou ajoutés en fin de RAG. [Haystack Documentation+1](#) [Haystack+1](#)
- Guide récapitulatif des métriques (context precision/recall, etc.). [Haystack](#)

5. Routage et fallbacks

- Exemples de **routing conditionnel** (par langue, par confiance) et **fallbacks websearch** — utiles pour bâtir des variantes robustes sans code ad-hoc. [Haystack](#)

6. Industrialisation minimale

- **Sérialisation** de pipelines (export/import) pour déploiement reproductible. [Haystack](#)

Ce qu'il vous restera à décider/intégrer

- Le **store** cible (p. ex. OpenSearch + BM25 et un vector store, ou Weaviate/Qdrant) selon vos prérequis on-prem. [Haystack Documentation](#)
- Les **modèles** (embeddings et **cross-encoder** de reranking) adaptés FR/EN et vos contraintes GPU/CPU. [Haystack](#)
- L'**orchestration serveur** (FastAPI/WSGI, auth SSO, journalisation) et l'intégration au SI.
- Les **règles métier** (filtres par juridiction/période, politiques de citations), qui se pluggent dans le pipeline.

En bref

Avec Haystack, vous avez déjà les **briques clés** d'un RAG hybride — ingestion, BM25 + dense, **fusion**, **reranking**, **évaluation** — et des **connecteurs document stores** pour rester évolutif et souverain on-prem. Vous composez les pipelines, choisissez vos backends et modèles, puis sérialisez/déployez sans recoder l'algorithme de retrieval. [Haystack Documentation+1](#)

Si vous le souhaitez, je vous propose un **squelette de dépôt** (Docker Compose) :

- Ingestion + indexation,
- Pipeline **hybride** (BM25 + dense + **bge-reranker**),
- Pipeline **d'évaluation** (faithfulness, context relevance, SAS).



produis moi un aperçu de ce que pourrait être la solution

Voici un **aperçu concret** d'une solution RAG hybride, "quasi clé en main", bâtie sur **Haystack** et déployable **on-prem**. L'objectif est d'obtenir une base robuste (BM25 + dense + reranking), souveraine, et **évolutive** vers KG/ontologie sans refonte.

1) Cibles et principes

- **Objectif** : questions/réponses fiables sur corpus internes (PDF, DOCX, emails exportés), avec **citations ancrées** (doc, page, paragraphe).
- **Architecture** : pipeline **hybride** (BM25 u dense) → **fusion** (RRF/pondérée) → **reranking cross-encoder** → génération contrôlée (LLM local 7–8B).
- **Évolutivité** : ajout ultérieur d'un **Graph-RAG** ciblé (obligations/acteurs/dates), puis d'une **couche ontologique** restreinte sur les domaines à ROI.

2) Architecture fonctionnelle (briques)

1. Ingestion & normalisation

- Connecteurs NAS/DMS, OCR (si scannés), extraction texte/layout, **découpe multi-granularité** (document → section → paragraphe → phrase).
- Enrichissement (métadonnées : type, parties, dates d'effet, version, langue).

2. Indexation

- **Lexical** (BM25) + **Dense** (embeddings FR/EN) dans un store compatible Haystack.
- Option A (intégrée "moteur") : **OpenSearch** (BM25 + dense/k-NN).
- Option B (DB vectorielle) : **Qdrant** pour dense, **OpenSearch** pour BM25.

3. Retrieval hybride

- Exécution parallèle BM25 et dense → **fusion** (RRF) → **top-K** candidats.

4. Reranking

- **Cross-encoder** (ex. bge-reranker-base) appliqué sur un **top-K restreint** (20–50).

5. Génération

- **LLM local 7-8B** (vLLM/TGI/llama.cpp) avec prompt “quote-first” + contraintes (langue, ton, style).

6. Citations & traçabilité

- Ancrage doc/page/§, score de confiance, journal d’audit (requête, versions).

7. Évaluation continue

- Recall@K, précision contextuelle, **faithfulness**, latence, taux de citations valides.

8. Sécurité

- SSO (Keycloak/LDAP), RBAC par espace documentaire, chiffrement au repos/en transit.

3) Interfaces I/O (contrats simplifiés)

3.1 Requête

```
json
{
  "query_id": "uuid",
  "text": "Quelle clause de résiliation s'applique au contrat X ?",
  "filters": {
    "type": ["contrat", "avenant"],
    "jurisdiction": ["FR"],
    "effective_at": "2024-06-01"
  },
  "lang": "fr"
}
```

3.2 Résultat

```
json
{
  "query_id": "uuid",
  "answer": "Selon l'avenant du 12/03/2024, ...",
  "citations": [
    {"doc_id": "contrat_X.pdf", "page": 17, "section": "4.2", "snippet": "... résiliation pour manquement ..."}
  ],
  "signals": {
    "dense_score": 0.78,
    "bm25_score": 12.3,
    "rerank_score": 0.91
  },
  "latency_ms": 820
}
```

4) Deux variantes d’implémentation

Variante A — OpenSearch-centrée (simple, très intégrée)

- **BM25 + dense** dans OpenSearch (index unique, champs denses/sparses).

- **Pipelines Haystack :**
BM25Retriever + DenseRetriever → DocumentJoiner(mode="reciprocal_rank_fusion") → CrossEncoderRanker → PromptNode(LLM local).
- **Avantages :** un seul cluster à exploiter, filtres/agrégations riches, scaling éprouvé.

Variante B — Qdrant + OpenSearch (vector-first)

- **Dense** dans Qdrant, **BM25** dans OpenSearch.
- **Haystack** orchestre la **double requête**, fusion, rerank, puis génération.
- **Avantages :** Qdrant très léger/rapide on-prem ; évolutif horizontalement.

5) Squelette minimal du pipeline Haystack (Python)

(extrait, noms indicatifs)

```
python

from haystack import Pipeline
from haystack.components.retrievers import BM25Retriever, DenseRetriever, DocumentJoiner
from haystack.components.rankers import TransformersCrossEncoderRanker
from haystack.components.builders import AnswerBuilder
from haystack_integrations import OpenSearchStore, QdrantStore

bm25 = BM25Retriever(document_store=OpenSearchStore(...))
dense = DenseRetriever(document_store=QdrantStore(...), embedding_model="bge-m3")
join = DocumentJoiner(strategy="reciprocal_rank_fusion")
rerank = TransformersCrossEncoderRanker(model="BAAI/bge-reranker-base", top_k=25)

rag = Pipeline()
rag.add_component("bm25", bm25)
rag.add_component("dense", dense)
rag.add_component("join", join)
rag.add_component("rerank", rerank)
# ... composant de génération branché sur un endpoint LLM local

rag.connect("bm25.documents", "join.documents_a")
rag.connect("dense.documents", "join.documents_b")
rag.connect("join.documents", "rerank.documents")
```

6) Déploiement on-prem (aperçu Compose)

(services essentiels ; à adapter selon votre variante)

```
yaml

version: "3.9"
services:
  opensearch:
    image: opensearchproject/opensearch:2.12.0
    environment: { discovery.type: single-node, ... }
    ulimits: { memlock: { soft: -1, hard: -1 } }
    ports: [ "9200:9200" ]

  qdrant:
    image: qdrant/qdrant:v1.11.0
    volumes: [ "qdrant:/qdrant/storage" ]
    ports: [ "6333:6333" ]

  llm:
```

```
image: vllm/vllm-openai
command: ["--model", "Qwen2-7B-Instruct", "--tensor-parallel-size", "1"]
ports: ["8000:8000"]
# GPU (si dispo): deploy.resources.reservations.devices...

reranker:
  image: ghcr.io/hf-inference/onnx-runtime
  # servez le cross-encoder via une petite API (FastAPI)

api:
  build: ./api # FastAPI: endpoints /ingest, /query, /eval
  environment: { LLM_BASE_URL: "http://llm:8000" }
  depends_on: [opensearch, qdrant, llm]
volumes: { qdrant: {} }
```

7) Endpoints applicatifs (FastAPI)

- **POST /ingest** : upload doc, OCR, split, embeddings, indexation (BM25 + dense), métadonnées.
- **POST /query** : payload "Requête" (cf. §3.1) → pipeline Haystack → résultat (cf. §3.2).
- **POST /evaluate** : lot de Q/A de test → métriques (recall@k, faithfulness, latence).
- **GET /health** : liveness/readiness ; **GET /metrics** : Prometheus.

8) Observabilité & gouvernance

- **Monitoring** : Prometheus + Grafana (latence, QPS, token/s, cache hits).
- **Audit** : journal des requêtes, empreintes/hashes des docs, versions indexées.
- **Sécurité** : Keycloak/LDAP (RBAC par collection/espace), TLS interne, chiffrement disque.

9) Dimensionnement indicatif

- **Atelier/équipe** : 1× GPU 24 Go (ou CPU quantisé), 16–32 vCPU, 128 Go RAM → LLM 7–8B + rerank top-25 < 1 s–2 s pour K=200.
- **Prod locale** : 2× GPU 24–48 Go, 32 vCPU, 256 Go RAM ; OpenSearch/Qdrant en réplication.

10) Évolutivité (roadmap)

- **V2** : multi-granularité stricte, filtres métier avancés (juridiction/période), "quote-first".
- **V3** : **mini Graph-RAG** ciblé (entités/obligations/pénalités) branché **après** le reranking pour contextualiser ; mêmes contrats I/O.
- **V4** : **ontologie restreinte** (JSON-LD) sur 3–5 catégories à forte valeur (Clause, Obligation, Pénalité, DonnéePersonnelle, Norme).

Résumé exécutif

- **Base recommandée** : Haystack + (OpenSearch **ou** OpenSearch+Qdrant) + **Cross-Encoder** + LLM 7–8B local.

- **Bénéfice** : pertinence élevée (hybride + rerank), souveraineté totale, **entièrement on-prem**, et trajectoire claire vers Graph/Onto sans casser l'existant.

Si vous le souhaitez, je peux transformer cet aperçu en **squelette de dépôt** prêt à cloner (arborescence, Compose, exemples d'appels d'API, pipeline Haystack sérialisé).



J'ai par ailleurs une autre réponse. Comment tout cela converge ou diverge ?

Pour aller le plus rapidement au but tout en restant évolutif, l'approche la plus pragmatique et directe consiste à combiner un framework d'orchestration RAG avec une base de données vectorielle légère et performante.

Basé sur notre discussion, voici la suggestion la plus efficace :

****Approche recommandée : Haystack + Qdrant****

Cette combinaison vous offre le meilleur rapport vitesse/flexibilité pour déployer une solution robuste chez vos clients.

1. Pourquoi c'est le plus rapide ?

* [cite_start]**Haystack agit comme un accélérateur "quasi clé en main"** : Il vous fournit directement les briques logicielles pour construire votre RAG sans réinventer l'algorithmique[cite: 711]. [cite_start]Vous pouvez composer des pipelines pour l'ingestion et la recherche hybride (BM25 + dense + reranker) en utilisant des composants pré-existants et des tutoriels clairs[cite: 684, 688].

* [cite_start]**Qdrant est simple à déployer et à gérer** : Contrairement à des solutions plus lourdes comme Elasticsearch, Qdrant peut être déployé via un simple binaire ou une image Docker, ce qui est idéal pour des environnements on-premise chez des clients variés[cite: 381, 569]. [cite_start]Il gère nativement la recherche hybride (dense + sparse/BM25), simplifiant encore l'architecture[cite: 567].

* [cite_start]**Moins de "colle" applicative** : Haystack gère la communication entre les différents composants (base de données, modèles d'embedding, reranker, LLM), ce qui vous fait gagner un temps de développement considérable[cite: 591, 613].

2. Pourquoi ça reste évolutif ?

* [cite_start]**Indépendance du stockage** : Haystack est conçu pour être agnostique vis-à-vis de la base de données[cite: 557, 691]. Vous pouvez commencer avec Qdrant pour sa simplicité et, si un client a des besoins spécifiques ou une infrastructure existante (comme OpenSearch), vous pouvez adapter le "backend" sans réécrire toute votre logique applicative.

* [cite_start]**Modularité de l'architecture** : Cette approche vous permet de commencer avec un RAG hybride performant, qui couvre 80% des besoins[cite: 377, 468]. Plus tard, vous pourrez ajouter des couches de complexité de manière ciblée, sans tout refondre :

* ****Ajout d'un Knowledge Graph (KG)**** : Le pipeline de retrieval reste le même. [cite_start]Vous pouvez simplement ajouter une nouvelle source de données (comme Neo4j) que Haystack interrogera en parallèle ou à la place du moteur vectoriel[cite: 607, 670].

* [cite_start]**Couche d'ontologie** : Une fois le RAG hybride stabilisé, vous pourrez l'enrichir

avec une ontologie métier pour améliorer l'explicabilité et la précision, en faisant de l'OntoGraph-RAG sélectif[cite: 394, 464].

Plan d'action concret pour aller vite

1. V1 - Le socle robuste (4-6 semaines) :

* [cite_start]Stack : Déployez une stack Docker Compose avec Qdrant, un service Haystack (via une API FastAPI), et un LLM 7-8B local (type Llama 3.1, servi par VLLM ou llama.cpp)[cite: 424, 425, 449].

* [cite_start]Pipeline : Créez un pipeline Haystack qui combine recherche lexicale (BM25) et vectorielle depuis Qdrant, suivi d'un reranker (type bge-reranker-base)[cite: 452, 686].

* Focus : Concentrez-vous sur la qualité de l'ingestion des documents (PDF, DOCX) et la pertinence des filtres de base (par exemple, dates, type de document).

2. V2 - La spécialisation métier :

* [cite_start]Une fois le socle validé, ajoutez des fonctionnalités spécifiques au domaine du client (par exemple, pour des juristes : le versioning des normes, la gestion de clauses contractuelles, des filtres sur les juridictions)[cite: 501, 505, 540].

3. V3 - L'avantage concurrentiel ciblé :

* [cite_start]Pour les clients les plus matures, introduisez un petit Graph-RAG sur un périmètre critique (par exemple, les obligations et pénalités dans les contrats) pour répondre à des questions complexes et relationnelles[cite: 456, 509, 541].

En résumé, l'approche Haystack + Qdrant vous permet de livrer rapidement de la valeur avec un RAG hybride performant, tout en vous garantissant une base saine et modulaire pour faire évoluer votre solution vers des architectures plus sophistiquées comme le Graph-RAG ou l'OntoGraph-RAG à l'avenir.

Merci. Voici une lecture « convergence / divergence » de cette proposition Haystack + Qdrant par rapport à ce que nous avons posé, avec une recommandation finale.

Convergences (alignement fort)

- **Approche pragmatique et rapide** : partir d'un **RAG hybride** (dense + lexical + rerank) avec un **framework d'orchestration** (Haystack) pour limiter le "code colle".
- **Évolutivité préservée** : Haystack est **agnostique du store** → on peut commencer sur Qdrant et, si besoin, basculer ou ajouter OpenSearch/Elasticsearch sans réécrire la logique métier.
- **Reranking cross-encoder** : appliqué sur un **top-K restreint**, c'est le bon compromis précision/latence.
- **Roadmap par paliers** : V1 (socle robuste) → V2 (spécialisation métier) → V3 (KG ciblé / OntoGraph sélectif).

Divergences / angles morts à traiter

1. BM25 dans Qdrant

- Qdrant propose désormais du **sparse/BM25** intégré, mais l'écosystème **lexical avancé** (analyzers multilingues, agrégations, filtres complexes, scoring par champs, highlighting) reste **meilleur** dans OpenSearch/Elasticsearch.
- Impact : pour des corpus juridiques/contrats riches en métadonnées et filtres stricts (juridiction, dates d'effet, versions), **OpenSearch** apporte plus de profondeur que le BM25 "léger" intégré à une vector-DB.

2. Filtres / agrégations "métier"

- Les besoins de **filtrage strict** (plages de dates effectives, catégories normatives, versions) et d'**agrégations** (comptages par type, période, BU) seront **nettement plus simples** et performants dans un **moteur de recherche** (OpenSearch/Elastic) que dans une pure vector-DB, même "hybride".

3. Observabilité et gouvernance

- OpenSearch/Elastic offrent nativement logs, audit, snapshots, ILM, RBAC fin, qui peuvent **accélérer la MCO on-prem**.
- Avec Qdrant-only, il faut **compléter** (Prometheus/Grafana, sauvegardes, politique de rétention), ce qui reste faisable mais un peu plus "à la carte".

Où ça converge bien (cas d'usage)

- **POC/atelier** ou **première prod contrôlée** (volumétrie modérée, peu d'agrégations) : **Haystack + Qdrant** est extrêmement efficace, minimal en opérations, excellent time-to-value.
- **Clients sans historique Elasticsearch** et cherchant un déploiement simple on-prem : Qdrant est un très bon point de départ.

Où ça peut diverger (et comment corriger)

- **Domaines très "lexicaux"** (juridique, conformité, régulation, documentation technique avec filtres complexes) :
 - Option 1 (simple) : **ajouter OpenSearch** pour BM25/filtres/agrégations et **garder Qdrant** pour le dense. Haystack orchestre les deux, puis **fusion + rerank**.
 - Option 2 (intégrée) : **OpenSearch-centré** (BM25 + k-NN) + rerank ONNX/Transformers → un seul cluster à exploiter.

Matrice de décision rapide

Contrainte dominante	Choix recommandé	Pourquoi
Déploiement très rapide, maintenance légère	Haystack + Qdrant	Simplicité opérationnelle, perf dense excellentes
Filtres juridiques complexes, agrégations, multi-langues	Haystack + OpenSearch (+ Qdrant si besoin)	BM25 avancé, analyzers, pipeline hybride natif
Plateforme "moteur de recherche" unique et très intégrée	OpenSearch-centré	Hybrid + RRF + ops/monitoring intégrés
API simple vector-first, scaling horizontal	Qdrant-centré	Léger, rapide, hybride possible, rerank côté app

Plan d'action pour concilier les deux approches

1. V1 (4-6 semaines)

- Implémenter **Haystack + Qdrant** (dense + sparse/BM25) + **cross-encoder**.
- Définir des **contrats I/O** stables (requête, candidats, citations) et une **API FastAPI**.
- Mettre en place **évaluation** (recall@k, context precision/recall, faithfulness) et **journalisation** (requête, versions doc).

2. V1.1 (switch optionnel, sans refonte)

- Brancher **OpenSearch** pour BM25 et filtres métier si besoin s'intensifie ; conserver Qdrant pour dense.
- Activer **fusion RRF** et garder le **rerank** sur top-K.

3. V2 (spécialisation métier)

- Ajout **versioning temporel** (effective_from/to), **bibliothèque de clauses**, gabarits de réponses "quote-first".

4. V3 (différenciation)

- **Mini KG** (obligations/acteurs/échéances) relié au pipeline post-rerank ; **ontologie restreinte** si besoin d'explicabilité accrue.

Recommandation finale

Votre proposition **Haystack + Qdrant** est **juste et efficace** pour aller vite tout en restant modulaire. Pour rester durable dans les contextes **fortement lexicaux et filtrés** (juridique, conformité), gardez **dès le départ l'option d'ajouter OpenSearch** en BM25/filtres, **sans changer le pipeline Haystack**. Autrement dit : **commencez Qdrant-only**, mais **cadrez l'interface** pour pouvoir passer en **duo OpenSearch+Qdrant** à la première montée en complexité, sans dette technique.



Faire un projet en s'appuyant sur les différents codes juridiques de la Nouvelle-Calédonie serait-il un bon projet ? Aurais-tu d'autres suggestions plus appropriées, sachant que mon but est d'apprendre, mais aussi de créer rapidement un produit livrable ?

Un projet basé sur les **codes juridiques de Nouvelle-Calédonie** peut être un excellent terrain d'apprentissage et de prototypage, mais il faut bien calibrer l'ambition et les contraintes :

Pourquoi c'est une bonne idée

- **Corpus bien délimité** : Les codes juridiques sont finis, publics, et structurés par livres, titres, articles. Faciles à collecter et maintenir.
- **Cas d'usage réel** : Beaucoup d'entreprises, juristes et administrations ont besoin de retrouver rapidement les bons articles, avec contexte, renvois, ou version en vigueur.
- **RAG hybride pertinent** : Le droit combine recherche lexicale (BM25 pour trouver par mots exacts) et recherche dense (trouver des articles proches sémantiquement). Ajouter un reranker améliore la

pertinence.

- **Potentiel pédagogique** : Cela couvre ingestion, indexation, pipeline RAG, UX de requête et réponse.

Limites à garder en tête

- **Maintenance** : Les codes changent (réformes, décrets), il faudra une stratégie d'actualisation.
- **Risque métier** : Produit sensible → il faut **avertir que ce n'est pas un conseil juridique**.
- **Compétition** : Des acteurs (Légifrance NC, bases juridiques) couvrent déjà ce besoin, même si pas toujours avec un RAG moderne.

Suggestions alternatives (même esprit, plus simple ou plus différenciant)

1. RAG sur conventions collectives et accords d'entreprise en NC

- Corpus limité, très utile aux DRH, syndicats, PME.
- Moins couvert que les codes → plus de valeur immédiate.

2. RAG sur procédures administratives locales (CAFAT, impôts, travail, CCI, etc.)

- Permet aux entreprises ou particuliers de comprendre leurs démarches.
- Donne un produit concret type "assistant démarches".

3. RAG technique pour le BTP (normes de sécurité, CCTP, référentiels techniques)

- Aligné avec votre étude actuelle sur les métiers du BTP en NC.
- Livrable directement utile aux entreprises industrielles.

4. Assistant documentaire interne pour entreprises clientes

- Ingestion de leurs **procédures internes, manuels, rapports** → assistant question-réponse interne.
- Produit livrable quasi immédiat, valeur ajoutée forte (gain de temps RH/qualité).

Recommandation pour apprendre + livrer vite

- **Prototype 1** : codes juridiques NC → pour apprendre la chaîne technique complète.
- **Produit livrable 1.1** : pivoter rapidement vers un corpus **spécifique aux entreprises locales** (conventions, normes BTP, procédures internes).
- Ainsi, vous capitalisez sur l'apprentissage tout en créant une version vendable sans trop d'obstacles juridiques.

Souhaitez-vous que je vous propose un **plan en 3 phases (prototype → MVP → produit livrable)** adapté à votre contexte ?



On m'a également proposé tout ceal. Fais moi une vue d'ensemble de tes suggestions et celles ci :

Autres suggestions plus appropriées pour un premier produit rapide

Si votre priorité absolue est la vitesse de livraison d'un produit robuste avec un minimum de complexité initiale, voici trois alternatives qui s'appuient sur la même architecture technique mais avec des corpus plus simples.

Projet 1 : L'assistant pour les publications officielles

Le Corpus : Les publications du Journal Officiel de la Nouvelle-Calédonie (JONC), les rapports de l'IEOM, de l'ISEE, ou les délibérations du Congrès et des provinces.

Pourquoi c'est approprié :

Moins risqué : L'enjeu est moins critique que le conseil juridique. Synthétiser un rapport économique ou retrouver une délibération est moins sensible qu'interpréter un article de loi.

Besoin réel : Les entreprises, consultants et journalistes ont besoin de veiller et de comprendre rapidement le contenu de ces publications denses.

Plus simple : La gestion de la temporalité est plus simple (basée sur la date de publication) et les interconnexions sont moins formelles que dans le droit.

Le Produit Livable : Une application web permettant de poser des questions sur les publications des 6 derniers mois. Par exemple : "Quelles ont été les dernières mesures votées concernant le secteur du BTP ?".

Projet 2 : Le RAG pour la base de connaissance d'une entreprise (le plus direct vers un produit commercial)

Le Corpus : La documentation interne d'une entreprise cliente en Nouvelle-Calédonie : procédures qualité, politiques RH, documentation technique, manuels de formation, etc.

Pourquoi c'est approprié :

Retour sur investissement (ROI) immédiat : C'est le cas d'usage RAG par excellence. Vous faites gagner un temps précieux aux employés et améliorez l'accès à l'information interne.

Périmètre maîtrisé : Vous travaillez sur un corpus de documents défini avec un premier client pilote.

Déploiement simple : L'architecture on-premise que nous avons définie (Docker, Qdrant, etc.) est parfaitement adaptée pour être déployée chez un client.

Le Produit Livable : Un chatbot interne pour une PME ou une direction d'une plus grande entreprise, capable de répondre aux questions des collaborateurs sur les procédures internes.

Projet 3 : L'expert du tourisme et de la culture calédonienne

Le Corpus : Guides touristiques (Lonely Planet, etc.), contenu de sites web institutionnels (Destination Province Nord/Sud/Iles), articles de blog, documents culturels sur l'histoire de la Nouvelle-Calédonie.

Pourquoi c'est approprié :

Faible risque : Le risque lié à une mauvaise réponse est quasi nul.

Attrayant et visible : C'est un projet plus "grand public" qui peut servir de belle vitrine technologique.

Créatif : Permet de travailler sur la qualité de la conversation et la personnalité du LLM.

Le Produit Livrable : Un chatbot ou une application web qui aide les touristes ou les résidents à planifier leurs activités. Exemples : "Je cherche une randonnée de 2 heures près de Bourail avec un point de vue, que me suggères-tu ?", "Explique-moi l'histoire du Bagne en quelques paragraphes".

Voici une vue d'ensemble synthétique des pistes que vous avez sur la table — mes suggestions initiales et celles qu'on vous a proposées — classées par **niveau de risque**, **pertinence métier**, et **rapidité de mise en œuvre**.

1. Approche juridique (vos premières idées)

- **Corpus** : Codes juridiques de Nouvelle-Calédonie, conventions collectives, accords d'entreprise.
- **Avantages** : Corpus clair et bien structuré, pertinence élevée pour juristes et RH, possibilité d'un produit différenciant.
- **Limites** : Risque métier (erreurs d'interprétation sensibles), maintenance complexe (textes évolutifs), forte concurrence (bases légales existantes).
- **Intérêt pédagogique** : Exercice complet d'un RAG hybride sur données exigeantes.
- **Potentiel livrable** : Assistant juridique interne (PME/industriel) mais avec disclaimer.

2. Approche institutionnelle (proposition externe – Projet 1)

- **Corpus** : JONC, rapports IEOM/ISEE, délibérations du Congrès/provinces.
- **Avantages** : Corpus public, sensible mais pas juridique, utile pour entreprises, consultants, journalistes. Temporalité simple (par date).
- **Limites** : Intérêt restreint au marché local, nécessité d'actualiser fréquemment.
- **Potentiel livrable** : Assistant de veille réglementaire/économique → « Quelles mesures récentes concernent le BTP ? ».

3. Approche documentaire interne (vos suggestions + proposition externe – Projet 2)

- **Corpus** : Documentation interne (procédures, politiques, manuels, etc.).
 - **Avantages** : ROI immédiat pour l'entreprise cliente, périmètre maîtrisé, adapté à votre contexte on-premise.
 - **Limites** : Nécessite un client pilote, moins « démonstratif » vers l'extérieur.
 - **Potentiel livrable** : Chatbot interne clé en main pour une PME ou une direction → assistant collaboratif.
-

4. Approche sectorielle (vos suggestions)

- **Corpus** : Normes et référentiels du BTP, procédures administratives locales.
 - **Avantages** : Aligné avec vos missions actuelles, forte valeur métier pour vos clients, niche peu couverte.
 - **Limites** : Peut demander une phase de collecte longue selon l'accessibilité des documents.
 - **Potentiel livrable** : Assistant technique pour le BTP (sécurité, CCTP, démarches administratives).
-

5. Approche grand public (proposition externe – Projet 3)

- **Corpus** : Guides touristiques, contenus institutionnels tourisme/culture, blogs.
 - **Avantages** : Risque quasi nul, projet vitrine attractif, créativité dans l'expérience conversationnelle.
 - **Limites** : Faible monétisation directe (plus une vitrine qu'un produit facturable).
 - **Potentiel livrable** : Chatbot touristique ou culturel → assistant grand public interactif.
-

Convergence & divergence

- **Convergence** : Tous les scénarios reposent sur le même socle technique (RAG hybride BM25 + embeddings + reranker sur Haystack + Qdrant). Vous pouvez passer d'un projet à l'autre avec le même pipeline, seul le corpus change.
 - **Divergence** : Le choix du corpus définit le niveau de risque métier, la vitesse de prototypage, et le potentiel de monétisation.
-

Vue stratégique

- **Pour apprendre rapidement** → Corpus juridique (codes NC) : plus exigeant, bon pour se former.
- **Pour livrer vite un client** → Documentation interne d'entreprise : ROI immédiat, déploiement réaliste on-premise.
- **Pour une vitrine publique** → Tourisme/culture : faible risque, démonstratif, bon pour communication.
- **Pour se positionner sur votre expertise locale** → BTP / normes administratives : plus long à assembler, mais très différenciant.

Voulez-vous que je vous propose un **scénario progressif** du type :
 Prototype (codes NC) → MVP (JONC / docs internes) → Produit différenciant (BTP / procédures locales) ?



Je vais combiner

1.D - Le Veilleur Stratégique (Analyse de l'information publique)

Concept : Un outil pour interroger et synthétiser les publications officielles (Journal Officiel, rapports ISEE/IEOM, délibérations).

Avantages : Moins risqué que le conseil juridique, utile pour la veille économique et stratégique.

Source : Ma suggestion.

2. Approche institutionnelle (proposition externe – Projet 1)

Corpus : JONC, rapports IEOM/ISEE, délibérations du Congrès/provinces.

Avantages : Corpus public, sensible mais pas juridique, utile pour entreprises, consultants, journalistes. Temporalité simple (par date).

Limites : Intérêt restreint au marché local, nécessité d'actualiser fréquemment.

Potentiel livrable : Assistant de veille réglementaire/économique → « Quelles mesures récentes concernent le BTP ? ».

le but est de fournir des informations compilées sur la nouvelle calédonie, avec une vision future d'enrichir avec des sources opendata de data.gouv.nc

C'est un très bon compromis. Votre combinaison entre **veille stratégique** (informations publiques) et **approche institutionnelle** (JONC, IEOM, ISEE, délibérations) donne un projet :

Projet : Le Veilleur Stratégique NC

But : Fournir aux entreprises, syndicats, institutions et consultants un assistant capable de synthétiser et contextualiser les informations publiques concernant la Nouvelle-Calédonie.

1. Sources initiales

- **Journal officiel de la Nouvelle-Calédonie (JONC)** : textes réglementaires, arrêtés, nominations.
- **Rapports de l'IEOM** : analyses économiques et financières.
- **Rapports de l'ISEE** : statistiques sociales, démographiques et économiques.
- **Délibérations du Congrès et des provinces** : orientation politique et sectorielle.

2. Extension future

- **Opendata.gouv.nc** : bases statistiques en libre accès (BTP, énergie, emploi, santé, etc.).
- **Communiqués institutionnels** : gouvernement, Haut-Commissariat, provinces.
- **Sources sectorielles** (BTP, énergie, nickel).

3. Fonctionnalités clés

- **Recherche avancée en langage naturel** : « Quelles mesures récentes concernent la fiscalité des entreprises ? ».
- **Synthèse thématique** : extraction de points clés par secteur (BTP, éducation, énergie).
- **Filtrage temporel** : par mois, trimestre, année.
- **Alertes automatisées** : push email / tableau de bord → « Nouveaux arrêtés sur la sécurité au travail cette semaine ».
- **Comparaison évolutive** : évolution d'un indicateur (emploi, énergie, importations).

4. Architecture (prototype réaliste avec Haystack)

- **Ingestion** : scrapers (PDF JONC, sites IEOM/ISEE), ingestion automatique.
- **Prétraitement** : extraction texte (pdfplumber, tika), segmentation par article/section.
- **Indexation** : Qdrant ou Weaviate (embeddings), BM25 (textes longs officiels).
- **RAG hybride** : combinaison retrieval lexical + sémantique, reranker.
- **Synthèse** : LLM (Mistral, Llama) pour répondre et produire résumés structurés.
- **Interface** :
 - Web minimaliste (FastAPI + HTML ou Streamlit)
 - Tableau de bord interactif (recherches, filtres, export PDF/CSV).

5. Livrable rapide

- **Phase 1 (MVP)** :
 - Corpus fixe (JONC + 1 rapport IEOM + 1 rapport ISEE).
 - Interface simple : recherche + réponse LLM.
- **Phase 2** :
 - Ajout filtres temporels, alertes email, opendata.
 - Tableau de bord comparatif par secteur.

6. Avantages

- **Apprentissage complet** : ingestion, prétraitement, RAG, synthèse.
- **Risques maîtrisés** : corpus public → pas de responsabilité juridique.
- **Pertinence locale forte** : cible entreprises, fédérations, institutions.
- **Scalabilité** : possibilité d'ajouter sources sans refondre la structure.

Ce projet est **rapide à prototyper** (vous pouvez avoir une démo en 4 à 6 semaines avec Haystack), **pertinent pour le marché NC**, et **évolutif** avec les opendata.

Voulez-vous que je vous esquisse un **MVP ultra-concret** (corpus limité, schéma d'architecture, interfaces minimales), que vous pourriez réellement coder en 2-3 semaines avec votre stack actuelle ?



OK. Et propose des noms pour le repo github, que je puisse le créer maintenant.
Je ne créerai pas de coker dans un premier temps.
Je continue avec ma logique actuelle.
Je switche sur haystack comm tu le proposes

""""

C'est un très bon compromis. Votre combinaison entre veille stratégique (informations publiques) et approche institutionnelle (JONC, IEOM, ISEE, délibérations) donne un projet :

Projet : Le Veilleur Stratégique NC

But : Fournir aux entreprises, syndicats, institutions et consultants un assistant capable de synthétiser et contextualiser les informations publiques concernant la Nouvelle-Calédonie.

1. Sources initiales

Journal officiel de la Nouvelle-Calédonie (JONC) : textes réglementaires, arrêtés, nominations.

Rapports de l'IEOM : analyses économiques et financières.

Rapports de l'ISEE : statistiques sociales, démographiques et économiques.

Délibérations du Congrès et des provinces : orientation politique et sectorielle.

2. Extension future

Opendata.gouv.nc : bases statistiques en libre accès (BTP, énergie, emploi, santé, etc.).

Communiqués institutionnels : gouvernement, Haut-Commissariat, provinces.

Sources sectorielles (BTP, énergie, nickel).

3. Fonctionnalités clés

Recherche avancée en langage naturel : « Quelles mesures récentes concernent la fiscalité des entreprises ? ».

Synthèse thématique : extraction de points clés par secteur (BTP, éducation, énergie).

Filtrage temporel : par mois, trimestre, année.

Alertes automatisées : push email / tableau de bord → « Nouveaux arrêtés sur la sécurité au travail cette semaine ».

Comparaison évolutive : évolution d'un indicateur (emploi, énergie, importations).

4. Architecture (prototype réaliste avec Haystack)

Ingestion : scrapers (PDF JONC, sites IEOM/ISEE), ingestion automatique.

Prétraitement : extraction texte (pdfplumber, tika), segmentation par article/section.

Indexation : Qdrant ou Weaviate (embeddings), BM25 (textes longs officiels).

RAG hybride : combinaison retrieval lexical + sémantique, reranker.

Synthèse : LLM (Mistral, Llama) pour répondre et produire résumés structurés.

Interface :

Web minimaliste (FastAPI + HTML ou Streamlit)

Tableau de bord interactif (recherches, filtres, export PDF/CSV).

5. Livrable rapide

Phase 1 (MVP) :

Corpus fixe (JONC + 1 rapport IEOM + 1 rapport ISEE).

Interface simple : recherche + réponse LLM.

Phase 2 :

Ajout filtres temporels, alertes email, opendata.

Tableau de bord comparatif par secteur.

6. Avantages

Apprentissage complet : ingestion, prétraitement, RAG, synthèse.

Risques maîtrisés : corpus public → pas de responsabilité juridique.

Pertinence locale forte : cible entreprises, fédérations, institutions.

Scalabilité : possibilité d'ajouter sources sans refondre la structure.

Ce projet est rapide à prototyper (vous pouvez avoir une démo en 4 à 6 semaines avec Haystack), pertinent pour le marché NC, et évolutif avec les opendata.

Voulez-vous que je vous esquisse un MVP ultra-concret (corpus limité, schéma d'architecture, interfaces minimales), que vous pourriez réellement coder en 2-3 semaines avec votre stack actuelle ?

""""

Voici une liste de noms pour ton dépôt GitHub, adaptés à ton projet de veille stratégique calédonienne avec Haystack :

Noms explicites (focus Nouvelle-Calédonie)

- `veille-strategique-nc`
- `rag-veille-nc`
- `veille-nc-rag`
- `veille-officielle-nc`
- `veille-institutionnelle-nc`

Noms orientés produit

- `nc-insight`
- `veilleur-nc`
- `nc-observatory`
- `observatoire-nc`
- `nc-veille-ai`

Noms techniques

- `haystack-nc-veille`
- `rag-hybride-nc`
- `nc-knowledge-rag`
- `nc-info-rag`

Noms sobres et professionnels

- `veille-strategique-nc-ai`
- `assistant-veille-nc`
- `nc-intelligence`
- `veille-data-nc`

Si tu veux mettre l'accent sur la **dimension locale + AI**, je recommande :

- **veille-strategique-nc** (clair, sobre, explicite)
- **nc-insight** (plus court, orienté produit)
- **veilleur-nc** (nom qui a du caractère et facile à retenir)

Veux-tu que je prépare aussi une **structure initiale du repo (README, dossiers ingestion / preprocessing / retriever / frontend)** pour que tu puisses le créer immédiatement et commencer à coder proprement ?



comment se dit veilleur dans les langues maohi, maori, polynésiennes etc ?

J'ai vérifié dans les langues polynésiennes proches (toutes issues de la famille austronésienne, avec racines partagées) :

En tahitien (reo mā'ohi, Polynésie française)

- **ti'a'i** : garder, protéger, veiller.
- **'āmura'a** : sentinelle, garde.
Donc « veilleur » = **te ti'a'i** (le gardien, celui qui veille).

En marquisien (Polynésie française)

- **taio** : surveiller, observer.
- « veilleur » peut se traduire par **te taio** (l'observateur).

En maori de Nouvelle-Zélande (te reo Māori)

- **kaitiaki** : gardien, protecteur, celui qui veille sur quelque chose (souvent utilisé pour les gardiens spirituels, environnementaux).
- **mātaki** : observer, regarder attentivement.
Donc « veilleur » = **kaitiaki** (le gardien).

En samoan

- **leoleo** : garde, policier, celui qui veille.
- **mata'itu** : surveiller, observer.
Donc « veilleur » = **leoleo**.

En hawaïen

- **kiai** : gardien, protecteur, veilleur.
- **maka'ala** : vigilant, alerte.
Donc « veilleur » = **kiai**.

Synthèse utilisable pour ton projet

Si tu veux un nom local fort mais lisible par un public francophone/anglophone :

- **Ti'a'i NC** (sobre, tahitien, très proche de la NC culturellement).
- **Kaitiaki NC** (maori, très connu dans le Pacifique, sens profond de gardien protecteur).
- **Kiai NC** (hawaïen, court et percutant).
- **Leoleo NC** (samoan, mais risque d'être confondu avec « Léo » en français).

Mon préféré pour toi : **Kaitiaki NC** → clair, polysémique (gardien/veilleur/protecteur), déjà connu dans le Pacifique, élégant comme nom de repo.

Veux-tu que je dresse une **short-list de 3 noms GitHub inspirés de ces langues + leur explication** pour que tu choisisses ?



Et dans les langues kanak ?

Je n'ai pas trouvé de traduction spécifique du terme « veilleur » dans les langues kanak, car ces langues sont majoritairement orales et peu documentées en ligne. Toutefois, je peux proposer des compilations informelles et des noms symboliques inspirés du contexte culturel kanak, tout en respectant les informations disponibles.

Contexte linguistique des langues kanak

- La Nouvelle-Calédonie compte une trentaine de langues kanak distinctes (Ajië, Drehu, Nengone, Paicî, Xârâcùù, numèè, etc.) [Wikipedia+1](#) [sorosoro.org+8](#).
- Ces langues sont surtout orales, et les ressources linguistiques disponibles en ligne se concentrent sur la classification ou les statistiques de locuteurs, plutôt que sur le vocabulaire fonctionnel [Wikipedia+1](#) [culture.gouv.fr](#).
- Aucun terme précis pour « veilleur », « surveillant » ou « gardien » n'est documenté dans les sources consultées.

Noms symboliques inspirés des langues kanak

Même en l'absence de traduction directe, vous pouvez créer un nom d'inspiration locale et symbolique. Voici quelques propositions :

Nom proposé	Langue (référence)	Signification / Raisons symboliques
Kaitiaki NC	Maori (Polynésie)	Gardien, protecteur — Terme largement reconnu dans le Pacifique
Ti'a'i NC	Tahitien (Océanie)	Le veilleur, le gardien
Kiai NC	Hawaïen	Veilleur, vigilant

Nom proposé	Langue (référence)	Signification / Raisons symboliques
Kaitiaki Kanak	Symbolique mélanéen	Gardien ; marie culture kanak et sens symbolique
Ti'a'i Ajië	Tahitien + référence relief Ajië	Gardien, inspiré d'une langue kanak connue
Kaitiaki Ajië	Māori + référence Ajië	Gardien, protecteur — lien symbolique avec le peuple Ajië

Recommandation

Étant donné le manque de vocabulaire validé dans les langues kanak, je propose de combiner un terme océanien (déjà utilisé dans le Pacifique) avec une référence culturelle ou linguistique locale. Par exemple :

- **Kaitiaki Kanak**
- **Ti'a'i Ajië**

Ces noms incarnent une symbolique forte (gardien/veilleur), tout en valorisant une référence culturelle locale. Ils offrent également une identité forte pour votre repo GitHub.

Souhaitez-vous que je vous propose également quelques variantes plus courtes ou accrocheuses (par ex. **Kananaki** ou **Tiaanaki**), inspirées de constructions mélanésiennes mais faciles à prononcer et mémoriser ?



Le concept s'appellera Kaitiaki.

esquisse un MVP ultra-concret (corpus limité, schéma d'architecture, interfaces minimales), que vous pourriez réellement coder en 2-3 semaines avec votre stack actuelle ?

Parfait. Voici une **esquisse de MVP "Kaitiaki"** réellement réalisable en 2–3 semaines avec votre stack actuelle, sans Docker, en local/on-prem, en basculant sur **Haystack + Qdrant** pour le RAG hybride.

1) Objectif et portée du MVP

- **But** : poser des questions en langage naturel et obtenir des réponses sourcées sur un corpus **limité** d'informations publiques récentes concernant la Nouvelle-Calédonie.
- **Corpus v1 (limité, statique)** :
 - JONC : 4–6 PDF des 3 derniers mois.
 - IEOM : 1 note de conjoncture récente.
 - ISEE : 1 publication statistique récente.
 - 1 jeu de délibérations (1 mois) du Congrès ou d'une Province.
- **Langues** : FR prioritaire, EN facultatif.

- **Livable** : une app web simple “Kaitiaki” avec champ de recherche, réponse synthétique + citations (doc, page, paragraphe), filtres temporels rudimentaires.

2) Architecture minimale locale

- **Backend** : Python 3.11, FastAPI.
- **Orchestration RAG** : Haystack 2.x (pipelines).
- **Stockage** :
 - BM25 : OpenSearch facultatif. Pour aller vite, commencez **Qdrant-only** avec sparse/BM25 intégré. Si besoin de filtres lexicaux avancés, vous ajouterez OpenSearch en V1.1.
 - Vecteurs : Qdrant local (service binaire ou simple process).
- **LLM** :
 - Local 7–8B (Llama 3.1 / Mistral 7B) via **llama.cpp** ou **vLLM** si vous avez un GPU.
 - Alternative immédiate si contrainte matérielle : petit modèle quantisé en CPU.
- **Reranker** : cross-encoder local (ex. **BAAI/bge-reranker-base**) via **sentence-transformers** ou composant Haystack.
- **Extraction PDF** : **pdfplumber** (+ Tesseract si scans).
- **Front minimal** : FastAPI + Jinja2 (SSR) ou Streamlit (si vous voulez aller encore plus vite).

3) Dépendances clés (requirements.txt)

```
makefile

fastapi==0.112.*
uvicorn[standard]==0.30.*
haystack-ai==2.*
qdrant-client==1.*
sentence-transformers==2.7.*
pdfplumber==0.11.*
pydantic==2.*
jinja2==3.*
numpy==1.26.*
```

Ajoutez **pytesseract** si OCR nécessaire, et le client de votre serveur LLM local (ex. **openai** si vous exposez vLLM en compat mode).

4) Arborescence du dépôt

```
bash

kaitiaki/
  README.md
  requirements.txt
  kaitiaki/
    __init__.py
    config/
      app.yaml          # chemins, modèles, top_k, etc.
    data/
      raw/              # PDFs bruts
      processed/        # textes extraits et JSON normalisés
```

```

ingest/
  fetch_sources.py      # copie locale / téléchargement manuel au début
  parse_pdf.py          # pdf -> texte, sections, pages
  normalize.py          # JSON {doc_id, title, date, text, sections, meta}
  indexer.py            # push BM25 + dense -> Qdrant (et/ou OpenSearch)
rag/
  pipeline.py           # Haystack: bm25 + dense -> fusion -> rerank
  llm_client.py         # wrapper vers LLM local
  schemas.py            # Pydantic DTO: Query, Hit, Answer, Citation
api/
  server.py             # FastAPI endpoints /query /ingest /health
  templates/
    index.html
    result.html
eval/
  eval_sets/            # 10-20 Q/A test
  evaluate.py           # recall@k, faithfulness (LLM judge)
utils/
  logging.py
  timeit.py

```

5) Config d'application (extrait config/app.yaml)

```

yaml

paths:
  data_raw: "kaitiaki/data/raw"
  data_processed: "kaitiaki/data/processed"

qdrant:
  host: "127.0.0.1"
  port: 6333
  collection: "kaitiaki_v1"
  use_sparse: true

models:
  embedding: "sentence-transformers/all-MiniLM-L6-v2"
  reranker: "BAAI/bge-reranker-base"
  llm_base_url: "http://127.0.0.1:8000/v1" # si vLLM compat OpenAI
  llm_model: "local/llama-3.1-8b-instruct"

retrieval:
  top_k_dense: 20
  top_k_bm25: 20
  fusion: "rrf"
  rerank_top_k: 25

ui:
  default_date_range_days: 180

```

6) Pipeline Haystack (extrait minimal)

```

python

# kaitiaki/rag/pipeline.py
from haystack import Pipeline
from haystack.components.retrievers import BM25Retriever, DenseRetriever, DocumentJoiner
from haystack.components.rankers import TransformersCrossEncoderRanker
from haystack.document_stores import QdrantDocumentStore
from sentence_transformers import SentenceTransformer

def build_pipeline(cfg):
    store = QdrantDocumentStore(host=cfg.qdrant.host, port=cfg.qdrant.port,
                               collection_name=cfg.qdrant.collection)

    # DENSE
    embedder = SentenceTransformer(cfg.models.embedding)
    dense = DenseRetriever(document_store=store, embedding_model=embedder)

```

```
# BM25 (sparse Qdrant) – si non disponible, remplacez par un retriever lexical maison
bm25 = BM25Retriever(document_store=store)
# Fusion
join = DocumentJoiner(strategy="reciprocal_rank_fusion")
# Rerank
rerank = TransformersCrossEncoderRanker(model=cfg.models.reranker,
top_k=cfg.retrieval.rerank_top_k)

pipe = Pipeline()
pipe.add_component("bm25", bm25)
pipe.add_component("dense", dense)
pipe.add_component("join", join)
pipe.add_component("rerank", rerank)

pipe.connect("bm25.documents", "join.documents_a")
pipe.connect("dense.documents", "join.documents_b")
pipe.connect("join.documents", "rerank.documents")
return pipe, store
```

7) Ingestion minimale

1. Déposez manuellement 6–10 PDF dans **data/raw**.
2. **parse_pdf.py** extrait texte + pages.
3. **normalize.py** produit un JSON normalisé par document :

```
json

{
  "doc_id": "jonc_2025-06-12.pdf",
  "title": "JONC - 12/06/2025",
  "date": "2025-06-12",
  "sections": [
    { "page": 4, "header": "Arrêté n°...", "text": "..." }
  ],
  "meta": { "source": "JONC", "uri": "file:///.../jonc_2025-06-12.pdf" }
}
```

4. **indexer.py** :

- crée des **documents Haystack** par **chunk** (ex. ~1200–1600 caractères), stocke **doc_id**, **page**, **section**, **date**.
- calcule embeddings denses et indexe dans Qdrant.
- si BM25 Qdrant indisponible, ajoutez un champ **sparse_vector** pré-calculé (BM25 maison) ou prévoyez un **plan B** OpenSearch en V1.1.

8) API minimale (FastAPI)

- **POST /ingest** : traite et indexe les PDF présents.
- **POST /query** :
 - Entrée : { "text": "...", "from": "2025-03-01", "to": "2025-08-25", "filters": {"source": ["JONC", "IEOM", "ISEE"]} }
 - Logique : pipeline → rerank → appel LLM → réponse + citations.
 - Sortie :

```
json

{
  "answer": "Synthèse ...",
  "citations": [
    {"doc_id": "junc_2025-06-12.pdf", "page": 4, "snippet": "..."},
    {"doc_id": "ieom_q2_2025.pdf", "page": 12, "snippet": "..."}
  ],
  "latency_ms": 820
}
```

- **GET /health** : statut.

9) Interface minimale

- Page unique SSR (Jinja2) :
 - Barre de recherche, sélecteur Sources (JONC/IEOM/ISEE/Délibérations), période (calendrier simple), bouton “Poser la question”.
 - Zone “Réponse” + 2-4 citations cliquables (ouvre un viewer PDF à la page).
 - Historique local des 10 dernières requêtes (localStorage).
- Variante Streamlit si vous voulez gagner encore du temps.

10) Scénario de test

- Questions types :
 - “Quelles mesures récentes concernent le BTP ?”
 - “Quels indicateurs IEOM sur l’activité économique au dernier trimestre ?”
 - “Quelles délibérations récentes sur l’énergie ?”
- Attendus :
 - 2-3 paragraphes de synthèse.
 - 2-4 citations exactes, page et extrait court.
 - Temps de réponse < 2 s si LLM local performant ; sinon 3-5 s acceptable en CPU quantisé.

11) Évaluation rapide (Semaine 2)

- **Retrieval** : recall@20 sur 10-15 Q/A internes (fait maison).
- **Pertinence** : note humaine 1-5.
- **Faithfulness** : LLM-as-judge local sur 10 réponses (déclaration de conformité aux citations).
- **Robustesse** : 50 requêtes aléatoires, mesurer latence moyenne/p95.

12) Sécurité et conformité de base

- Mention claire : “Kaitiaki fournit des synthèses d’informations publiques. Ce n’est pas un avis juridique.”

- Journalisation locale des requêtes (horodatage, IP locale si pertinent).
- Hash des documents indexés pour la traçabilité.

13) Plan en 2-3 semaines

- **Jour 1-3** : mise en place env Python, Qdrant local, FastAPI squelette, page SSR.
- **Jour 4-7** : ingestion PDF → texte → normalisation → indexation dense + sparse.
- **Jour 8-11** : pipeline Haystack (bm25 + dense → RRF → rerank), intégration LLM local, endpoint `/query`.
- **Jour 12-14** : UI, citations cliquables, tests de charge, lot Q/A d'évaluation, ajustements.
- **Option V1.1** : ajout OpenSearch si vous avez besoin de filtres BM25 avancés (sans casser l'API).

14) Évolutions prévues

- Filtres métier par thème (énergie, BTP, emploi) via taxonomie légère.
- Alertes email hebdomadaires : "Nouveautés JONC sur BTP".
- Intégration jeux **OpenData** (data.gouv.nc) : indicateurs avec mini-graphiques, en gardant le même pipeline.

Résumé exécutable

- **Repo** : `kaitiaki` (ou `kaitiaki-nc`).
- **Stack** : FastAPI + Haystack 2.x + Qdrant + LLM 7-8B local + reranker bge.
- **MVP** : un champ de recherche, réponses synthétiques, citations précises, filtres simples.
- **Sans Docker** : venv Python, Qdrant lancé en local, LLM local via `llama.cpp` ou `vLLM`.

Si vous le souhaitez, je peux fournir immédiatement :

- un **README.md** de démarrage,
- un **fichier requirements.txt** finalisé,
- un **script indexer.py** minimal et le **pipeline Haystack** complet,
- un **gabarit FastAPI** avec les endpoints présentés.



fournis les éléments suivants

Très bien. Voici un **kit de départ complet** pour le MVP « Kaitiaki » (sans Docker), prêt à copier-coller dans un dépôt GitHub. Il suit la structure annoncée et reste minimaliste tout en étant réellement exécutable.

Arborescence

```
bash

kaitiaki/
  README.md
  requirements.txt
  kaitiaki/
    config/app.yaml
    data/raw/          # vos PDFs à déposer ici
    data/processed/    # textes/JSON générés
    ingest/
      parse_pdf.py
      normalize.py
      indexer.py
    rag/
      pipeline.py
      llm_client.py
      schemas.py
      fusion.py
    api/
      server.py
      templates/
        index.html
        result.html
    utils/
      logging.py
      text.py
```

1) README.md

```
markdown

# Kaitiaki – Veille stratégique NC (MVP)

Assistant de veille pour interroger et synthétiser des publications publiques (JONC, IEOM, ISEE, délibérations).

## Objectif MVP
- Corpus limité (6–10 PDF récents).
- RAG hybride **sans Docker** :
  - dense (Qdrant + embeddings),
  - lexical (BM25 local via `rank-bm25`),
  - fusion RRF,
  - reranking (cross-encoder),
  - génération via LLM local (7–8B recommandé).

## Prérequis
- macOS / Linux / Windows
- Python 3.11+
- Qdrant **binaire** (server) lancé localement (par défaut : `127.0.0.1:6333`)
- Modèle LLM local (ex. Llama 3.1 / Mistral) servi en API compatible OpenAI (ex. `vLLM`) **ou** `llama.cpp` + petit wrapper HTTP.

## Installation
```bash
python -m venv .venv && source .venv/bin/activate # Windows: .venv\Scripts\activate
pip install -r requirements.txt
```

## Config

Éditez `kaitiaki/config/app.yaml` :

- chemins,
- nom de collection Qdrant,
- modèles (embedding, reranker, LLM),
- paramètres retrieval.

## Données

Déposez 6–10 PDF dans `kaitiaki/data/raw/`.

## Pipeline d'ingestion

```
bash

python -m kaitiaki.ingest.parse_pdf
python -m kaitiaki.ingest.normalize
python -m kaitiaki.ingest.indexer
```

## Lancer l'API

```
bash

uvicorn kaitiaki.api.server:app --reload
```

Ouvrez <http://127.0.0.1:8000> ↗

## Endpoints

- **POST** `/ingest` : (ré)indexation des fichiers présents.
- **POST** `/query` : `{ "text": "...", "from": "2025-05-01", "to": "2025-08-25", "sources": ["JONC", "IEOM", "ISEE"] }`
- **GET** `/health`

## Notes

- BM25 est implémenté localement via **rank-bm25** (pickle). Pour des filtres lexicaux avancés, vous pourrez **ajouter OpenSearch** en V1.1 sans changer l'API.
- Le LLM local doit exposer une route OpenAI-compatible (`/v1/chat/completions`).

## Licence / Avertissement

Kaitiaki fournit des synthèses d'informations publiques. **Ce n'est pas un avis juridique.**

```
yaml

```

```
2) requirements.txt

```txt
fastapi==0.112.2
uvicorn[standard]==0.30.6

haystack-ai==2.2.1
haystack-bm25==1.0.0

# Intégration Qdrant (document store)
haystack-ai[document-stores-qdrant]==2.2.1
qdrant-client==1.9.2

sentence-transformers==2.7.0
transformers==4.43.4
torch>=2.1.0

pdfplumber==0.11.4
pydantic==2.8.2
pyyaml==6.0.2

rank-bm25==0.2.2

jinja2==3.1.4

numpy==1.26.4
scikit-learn==1.5.1
```

3) config/app.yaml

```
yaml

paths:
  data_raw: "kaitiaki/data/raw"
  data_processed: "kaitiaki/data/processed"
  bm25_index: "kaitiaki/data/processed/bm25_index.pkl"
  bm25_meta: "kaitiaki/data/processed/bm25_meta.json"

qdrant:
  host: "127.0.0.1"
  port: 6333
  collection: "kaitiaki_v1"

models:
  embedding: "sentence-transformers/all-MiniLM-L6-v2"
  reranker: "BAAI/bge-reranker-base"
  llm_base_url: "http://127.0.0.1:8000/v1" # OpenAI-like endpoint (vLLM/serveur maison)
  llm_model: "local/llama-3.1-8b-instruct"
  llm_max_ctx: 4096

retrieval:
  top_k_dense: 20
  top_k_bm25: 20
  fusion: "rrf"
  rerank_top_k: 25

ui:
  default_days: 180
```

4) ingest/parse_pdf.py

python

```
# kaitiaki/ingest/parse_pdf.py
from pathlib import Path
import pdfplumber
import json
import os

RAW = Path(__file__).resolve().parents[1] / "data" / "raw"
PROC = Path(__file__).resolve().parents[1] / "data" / "processed"
PROC.mkdir(parents=True, exist_ok=True)

def extract_text(pdf_path: Path):
    pages = []
    with pdfplumber.open(str(pdf_path)) as pdf:
        for i, page in enumerate(pdf.pages, start=1):
            text = page.extract_text() or ""
            pages.append({"page": i, "text": text})
    return pages

def main():
    for f in sorted(RAW.glob("*.pdf")):
        out = PROC / f"{f.stem}.pages.json"
        if out.exists():
            continue
        pages = extract_text(f)
        with open(out, "w", encoding="utf-8") as w:
            json.dump({"doc_id": f.name, "pages": pages}, w, ensure_ascii=False, indent=2)
        print("Parsed:", f.name)

if __name__ == "__main__":
    main()
```

5) ingest/normalize.py

python

```
# kaitiaki/ingest/normalize.py
from pathlib import Path
import json
import re
from datetime import datetime

PROC = Path(__file__).resolve().parents[1] / "data" / "processed"

def guess_date_from_filename(name: str) -> str:
    m = re.search(r"(20\d{2}[-_]?d{2}[-_]?d{2})", name)
    if not m:
        return None
    s = m.group(1).replace("_", "-")
    if len(s) == 8: # yyyymmdd
        s = f"{s[:4]}-{s[4:6]}-{s[6:]}"
    return s

def chunk_text(text: str, size=1400, overlap=200):
    text = re.sub(r"\s+", " ", text).strip()
    chunks = []
    start = 0
    while start < len(text):
        end = min(len(text), start + size)
        chunks.append(text[start:end])
        start = end - overlap
        if start < 0: start = 0
    return chunks

def main():
    for f in sorted(PROC.glob("*.pages.json")):
        data = json.loads(f.read_text(encoding="utf-8"))
```

```

doc_id = data["doc_id"]
date = guess_date_from_filename(doc_id) or datetime.today().date().isoformat()

chunks = []
for p in data["pages"]:
    for c in chunk_text(p["text"] or ""):
        chunks.append({
            "doc_id": doc_id,
            "date": date,
            "page": p["page"],
            "text": c,
            "source": "AUTO", # JONC/IEOM/ISEE à ajuster selon le fichier
        })

    out = PROC / f"{Path(doc_id).stem}.normalized.json"
    out.write_text(json.dumps({"doc_id": doc_id, "date": date, "chunks": chunks},
ensure_ascii=False, indent=2), encoding="utf-8")
    print("Normalized:", doc_id, f"({len(chunks)} chunks)")

if __name__ == "__main__":
    main()

```

6) ingest/indexer.py

```

python

# kaitiaki/ingest/indexer.py
from pathlib import Path
import json
import pickle
from collections import defaultdict

import yaml
from rank_bm25 import BM25Okapi
from sentence_transformers import SentenceTransformer

from haystack import Document
from haystack_integrations.document_stores.qdrant import QdrantDocumentStore

CFG = yaml.safe_load((Path(__file__).resolve().parents[1] / "config" /
"app.yaml").read_text(encoding="utf-8"))
PROC = Path(__file__).resolve().parents[1] / "data" / "processed"

def yield_chunks():
    for f in sorted(PROC.glob("*.normalized.json")):
        data = json.loads(f.read_text(encoding="utf-8"))
        for ch in data["chunks"]:
            yield ch

def build_bm25_index(chunks):
    # Corpus tokenization simple (FR/EN)
    def tok(s): return [t for t in s.lower().split() if len(t) > 2]
    texts = [c["text"] for c in chunks]
    tokenized = [tok(t) for t in texts]
    bm25 = BM25Okapi(tokenized)
    meta = [{"doc_id": c["doc_id"], "page": c["page"]} for c in chunks]
    return bm25, tokenized, meta

def main():
    # 1) Qdrant DocumentStore
    store = QdrantDocumentStore(
        host=CFG["qdrant"]["host"],
        port=CFG["qdrant"]["port"],
        collection_name=CFG["qdrant"]["collection"],
        embedding_dim=384, # all-MiniLM-L6-v2
        recreate_index=True,
    )

    # 2) Charger et écrire les documents (chunks)
    docs = []

```

```

for ch in yield_chunks():
    d = Document(
        content=ch["text"],
        meta={"doc_id": ch["doc_id"], "page": ch["page"], "date": ch["date"], "source":
ch.get("source", "AUTO")}
    )
    docs.append(d)

if not docs:
    print("Aucun document à indexer. Vérifiez data/processed.")
    return

store.write_documents(docs)
print(f"Écrits dans Qdrant: {len(docs)} chunks")

# 3) Embeddings (dense)
embedder = SentenceTransformer(CFG["models"]["embedding"])
embs = embedder.encode([d.content for d in docs], batch_size=64, normalize_embeddings=True,
show_progress_bar=True)
# Sauvegarde des embeddings dans le store
for doc, emb in zip(docs, embs):
    doc.embedding = emb.tolist()
store.update_embeddings(docs)
print("Embeddings mis à jour.")

# 4) BM25 local (pickle)
bm25, tokenized, meta = build_bm25_index([{"text": d.content, **d.meta} for d in docs])
with open(CFG["paths"]["bm25_index"], "wb") as w:
    pickle.dump({"bm25": bm25, "tokenized": tokenized, w)
Path(CFG["paths"]["bm25_meta"]).write_text(json.dumps(meta, ensure_ascii=False, indent=2),
encoding="utf-8")
print("Index BM25 local créé.")

if __name__ == "__main__":
    main()

```

7) rag/schemas.py

```

python

# kaitiaki/rag/schemas.py
from pydantic import BaseModel, Field
from typing import List, Optional

class Query(BaseModel):
    text: str
    sources: Optional[List[str]] = None
    date_from: Optional[str] = None
    date_to: Optional[str] = None
    top_k: int = 20

class Citation(BaseModel):
    doc_id: str
    page: int
    snippet: str

class Answer(BaseModel):
    answer: str
    citations: List[Citation] = Field(default_factory=list)
    latency_ms: int = 0

```

8) rag/fusion.py (RRF)

python

```
# kaitiaki/rag/fusion.py
def rrf_merge(list_a, list_b, k=60):
    """Reciprocal Rank Fusion: chaque liste est une liste de tuples (id, score) ou (id, obj).
    On renvoie une liste d'ids ordonnés par score RRF.
    """
    ranks = {}
    for lst in [list_a, list_b]:
        for rank, (id_, _) in enumerate(lst, start=1):
            ranks[id_] = ranks.get(id_, 0.0) + 1.0 / (k + rank)
    return sorted(ranks.items(), key=lambda x: x[1], reverse=True)
```

9) rag/llm_client.py

python

```
# kaitiaki/rag/llm_client.py
import requests
import yaml
from pathlib import Path

CFG = yaml.safe_load((Path(__file__).resolve().parents[1] / "config" /
"app.yaml").read_text(encoding="utf-8"))

def generate_answer(question: str, contexts: list[str]) -> str:
    """Appel à une API OpenAI-compatible locale (vLLM ou autre)."""
    prompt = (
        "Vous êtes Kaitiaki, assistant de veille. "
        "Répondez en français de manière concise et sourcée. "
        "Citez les extraits pertinents en fin de réponse.\n\n"
        f"Question: {question}\n\n"
        "Contexte:\n" + "\n--\n".join(contexts[:8]) + "\n\n"
        "Réponse:"
    )

    body = {
        "model": CFG["models"]["llm_model"],
        "messages": [{"role": "user", "content": prompt}],
        "temperature": 0.2,
        "max_tokens": 600,
    }
    r = requests.post(f'{CFG["models"]["llm_base_url"].rstrip("/")}/chat/completions',
    json=body, timeout=120)
    r.raise_for_status()
    return r.json()["choices"][0]["message"]["content"].strip()
```

10) rag/pipeline.py (hybride + rerank)

python

```
# kaitiaki/rag/pipeline.py
import time
import json
import pickle
from pathlib import Path
from typing import List, Tuple

import yaml
import numpy as np
from sentence_transformers import SentenceTransformer, CrossEncoder
from haystack import Document
from haystack_integrations.document_stores.qdrant import QdrantDocumentStore
```

```

from .fusion import rrf_merge

CFG = yaml.safe_load((Path(__file__).resolve().parents[1] / "config" /
"app.yaml").read_text(encoding="utf-8"))

def _load_bm25():
    with open(CFG["paths"]["bm25_index"], "rb") as f:
        pk = pickle.load(f)
    meta = json.loads(Path(CFG["paths"]["bm25_meta"]).read_text(encoding="utf-8"))
    return pk["bm25"], pk["tokenized"], meta

def _bm25_search(query: str, top_k: int) -> List[Tuple[int, float]]:
    bm25, tokenized, meta = _load_bm25()
    toks = [t for t in query.lower().split() if len(t) > 2]
    scores = bm25.get_scores(toks)
    idx = np.argsort(scores)[::-1][:top_k]
    return [(int(i), float(scores[i])) for i in idx]

def _dense_search(store: QdrantDocumentStore, embedder: SentenceTransformer, query: str, top_k:
int):
    q = embedder.encode([query], normalize_embeddings=True)[0]
    docs: List[Document] = store.query_by_embedding(q, top_k=top_k)
    return docs

def hybrid_search(query: str, top_k_dense=20, top_k_bm25=20, rerank_top_k=25):
    t0 = time.time()
    # 1) stores / modèles
    store = QdrantDocumentStore(
        host=CFG["qdrant"]["host"],
        port=CFG["qdrant"]["port"],
        collection_name=CFG["qdrant"]["collection"],
        embedding_dim=384,
    )
    embedder = SentenceTransformer(CFG["models"]["embedding"])
    reranker = CrossEncoder(CFG["models"]["reranker"])

    # 2) dense + bm25
    dense_docs = _dense_search(store, embedder, query, top_k_dense)
    bm25_list = _bm25_search(query, top_k_bm25) # [(idx, score)]
    bm25_meta = json.loads(Path(CFG["paths"]["bm25_meta"]).read_text(encoding="utf-8"))

    # Pour BM25: on ne stocke pas le texte; on reconstruit des "pseudo-docs" minimalistes
    bm25_docs = []
    for i, sc in bm25_list:
        m = bm25_meta[i]
        # Récupérer le chunk depuis Qdrant par filtre meta (doc_id + page)
        hits = store.filter_documents(filters={"doc_id": m["doc_id"], "page": m["page"]})
        if hits:
            bm25_docs.append(hits[0])

    # 3) Fusion RRF (sur identifiants)
    def key(d: Document):
        return f'{d.meta.get("doc_id")}#p{d.meta.get("page")}#{hash(d.content)}'

    dense_pairs = [(key(d), 1.0) for d in dense_docs]
    bm25_pairs = [(key(d), 1.0) for d in bm25_docs]
    fused_ids = [k for k, _ in rrf_merge(dense_pairs, bm25_pairs)]

    # Limiter pour reranking
    id_to_doc = {key(d): d for d in (dense_docs + bm25_docs)}
    candidates = [id_to_doc[i] for i in fused_ids if i in id_to_doc][:rerank_top_k]

    # 4) Rerank (cross-encoder)
    pairs = [(query, d.content) for d in candidates]
    scores = reranker.predict(pairs)
    ranked = sorted(zip(candidates, scores), key=lambda x: x[1], reverse=True)

    latency = int((time.time() - t0) * 1000)
    return ranked, latency

```


11) api/server.py (FastAPI + templates)

```
python

# kaitiaki/api/server.py
from fastapi import FastAPI, Request, Body
from fastapi.responses import HTMLResponse, JSONResponse
from fastapi.templating import Jinja2Templates
from pathlib import Path
import yaml
import time

from kaitiaki.rag.pipeline import hybrid_search
from kaitiaki.rag.llm_client import generate_answer
from kaitiaki.rag.schemas import Query, Answer

app = FastAPI()
templates = Jinja2Templates(directory=Path(__file__).parent / "templates")

CFG = yaml.safe_load((Path(__file__).resolve().parents[1] / "config" /
"app.yaml").read_text(encoding="utf-8"))

@app.get("/health")
def health():
    return {"ok": True}

@app.get("/", response_class=HTMLResponse)
def home(request: Request):
    return templates.TemplateResponse("index.html", {"request": request})

@app.post("/ingest")
def ingest():
    # Pour MVP: lancez parse -> normalize -> indexer avec des sous-process si nécessaire
    return {"status": "TODO (lancer parse/normalize/indexer en local)"}

@app.post("/query", response_model=Answer)
def query(q: Query = Body(...)):
    ranked, rt = hybrid_search(q.text, rerank_top_k= min(q.top_k, CFG["retrieval"]
["rerank_top_k"]))
    # Concaténer quelques contextes
    contexts = [d.content for d, _ in ranked[:8]]
    t0 = time.time()
    answer_text = generate_answer(q.text, contexts)
    gen_ms = int((time.time() - t0) * 1000)

    # Citations simples (top 3)
    cits = []
    for d, _ in ranked[:3]:
        cits.append({
            "doc_id": d.meta.get("doc_id"),
            "page": d.meta.get("page"),
            "snippet": (d.content[:240] + "...") if len(d.content) > 240 else d.content
        })

    return Answer(answer=answer_text, citations=cits, latency_ms=rt + gen_ms)

# Pages simples
@app.post("/search", response_class=HTMLResponse)
def search_page(request: Request, text: str = Body(..., embed=True)):
    ranked, rt = hybrid_search(text)
    contexts = [d.content for d, _ in ranked[:6]]
    answer = generate_answer(text, contexts)
    cits = [{"doc_id": d.meta.get("doc_id"), "page": d.meta.get("page")} for d, _ in ranked[:3]]
    return templates.TemplateResponse("result.html", {"request": request, "q": text, "answer":
answer, "cits": cits})
```

12) api/templates/index.html

```
html

<!doctype html>
<html>
<head>
  <meta charset="utf-8">
  <title>Kaitiaki – Veille NC</title>
  <style>
    body { font-family: system-ui, sans-serif; max-width: 880px; margin: 2rem auto; }
    input, button { font-size: 1rem; padding: .6rem; }
    .row { display: flex; gap: .5rem; }
  </style>
</head>
<body>
  <h1>Kaitiaki – Veille NC (MVP)</h1>
  <form method="post" action="/search">
    <div class="row">
      <input name="text" placeholder="Ex: Quelles mesures récentes concernant le BTP ?"
style="flex:1" />
      <button type="submit">Chercher</button>
    </div>
  </form>
</body>
</html>
```

13) api/templates/result.html

```
html

<!doctype html>
<html>
<head>
  <meta charset="utf-8">
  <title>Résultat – Kaitiaki</title>
  <style>
    body { font-family: system-ui, sans-serif; max-width: 880px; margin: 2rem auto; }
    .cit { font-size: .92rem; color: #333; margin: .4rem 0; }
  </style>
</head>
<body>
  <h2>Question</h2>
  <p>{{ q }}</p>
  <h2>Réponse</h2>
  <pre style="white-space: pre-wrap">{{ answer }}</pre>
  <h3>Citations</h3>
  {% for c in cits %}
    <div class="cit">• {{ c.doc_id }} – page {{ c.page }}</div>
  {% endfor %}
  <p><a href="/">Nouvelle recherche</a></p>
</body>
</html>
```

14) utils/logging.py (facultatif)

```
python

import logging

def setup_logger(name="kaitiaki", level=logging.INFO):
    logger = logging.getLogger(name)
    if not logger.handlers:
        h = logging.StreamHandler()
        fmt = logging.Formatter("%(asctime)s %(levelname)s %(name)s: %(message)s")
```

```
h.setFormatter(fmt)
logger.addHandler(h)
logger.setLevel(level)
return logger
```

15) utils/text.py (facultatif)

```
python

import re

def normalize_ws(s: str) -> str:
    return re.sub(r"\s+", " ", s).strip()
```

Remarques finales

- **Sans Docker** : Qdrant doit être lancé via son **binaire** local (port 6333 par défaut). Conservez ce choix pour le MVP ; vous pourrez conteneuriser plus tard si nécessaire.
- **BM25** : l'index BM25 est local (pickle) pour aller vite. En V1.1, vous pourrez **brancher OpenSearch** pour bénéficier d'analyzers, filtres par champs et agrégations — **sans changer l'API**.
- **LLM local** : adaptez `models.llm_base_url` et `models.llm_model` à votre serveur LLM. Le wrapper `llm_client.py` suppose une API *OpenAI-like*.

Si vous le souhaitez, je peux ajouter un **script de tests** (10–15 Q/A) et une **checklist** de validations (latence p95, recall@20, taux de citations valides) pour boucler votre V1 en conditions réelles.



ajouter un script de tests (10–15 Q/A) et une checklist de validations (latence p95, recall@20, taux de citations valides) pour boucler votre V1 en conditions réelles

Parfait. Voici un **kit d'évaluation** prêt à ajouter à votre dépôt, comprenant :

1. un **jeu de tests** (10–15 Q/A) à personnaliser,
2. un **script d'évaluation** mesurant **latence p95**, **recall@20**, et **taux de citations valides**,
3. une **checklist** de validation V1.

1) Fichiers à ajouter

```
bash

kaitiaki/
  eval/
    qa_testset.json
    evaluate.py
```

eval/qa_testset.json (exemple à adapter)

Remplacez les **doc_id**/page par ceux présents dans **data/processed/*.pages.json**.

```
json
[
  {
    "question": "Quelles mesures récentes concernent le BTP ?",
    "expected_citations": [
      {"doc_id": "jonc_2025-06-12.pdf", "page": 4},
      {"doc_id": "congres_deliberations_2025-06.pdf", "page": 12}
    ]
  },
  {
    "question": "Quels sont les derniers indicateurs IEOM sur l'activité économique ?",
    "expected_citations": [
      {"doc_id": "ieom_conjoncture_T2_2025.pdf", "page": 8}
    ]
  },
  {
    "question": "Quelles actualités récentes de l'ISEE sur l'emploi ?",
    "expected_citations": [
      {"doc_id": "isee_indicateurs_emploi_2025-05.pdf", "page": 3}
    ]
  },
  {
    "question": "Quelles décisions récentes concernent l'énergie ?",
    "expected_citations": [
      {"doc_id": "congres_deliberations_2025-06.pdf", "page": 7}
    ]
  },
  {
    "question": "Quelles évolutions budgétaires ont été publiées au JONC ce mois-ci ?",
    "expected_citations": [
      {"doc_id": "jonc_2025-07-03.pdf", "page": 2}
    ]
  },
  {
    "question": "Quelles statistiques démographiques ont été publiées récemment par l'ISEE ?",
    "expected_citations": [
      {"doc_id": "isee_population_2025-04.pdf", "page": 5}
    ]
  },
  {
    "question": "Quelles mesures récentes en matière de sécurité au travail ?",
    "expected_citations": [
      {"doc_id": "jonc_2025-06-26.pdf", "page": 10}
    ]
  },
  {
    "question": "Quelles orientations récentes pour les infrastructures ?",
    "expected_citations": [
      {"doc_id": "congres_deliberations_2025-05.pdf", "page": 9}
    ]
  },
  {
    "question": "Quels constats IEOM sur le crédit aux entreprises ?",
    "expected_citations": [
      {"doc_id": "ieom_conjoncture_T1_2025.pdf", "page": 6}
    ]
  },
  {
    "question": "Quelles publications ISEE sur l'énergie et la consommation ?",
    "expected_citations": [
      {"doc_id": "isee_energie_2025-03.pdf", "page": 4}
    ]
  },
  {
    "question": "Quels textes JONC sur l'environnement ont été publiés récemment ?",
    "expected_citations": [
      {"doc_id": "jonc_2025-05-15.pdf", "page": 11}
    ]
  },
  {
  }
```

```

"question": "Quelles informations sur les importations récentes ?",
"expected_citations": [
  {"doc_id": "isee_commerce_exterieur_2025-06.pdf", "page": 2}
],
},
{
  "question": "Quelles dispositions récentes touchent l'éducation ?",
  "expected_citations": [
    {"doc_id": "congres_deliberations_2025-04.pdf", "page": 5}
  ],
},
{
  "question": "Quelles annonces JONC sur la fiscalité des entreprises ?",
  "expected_citations": [
    {"doc_id": "jonc_2025-07-10.pdf", "page": 3}
  ],
},
{
  "question": "Quelles mesures récentes concernant le tourisme ?",
  "expected_citations": [
    {"doc_id": "congres_deliberations_2025-06.pdf", "page": 15}
  ],
}
]

```

eval/evaluate.py

Mesure : **p95 latence** (end-to-end via /query), **recall@20** (sur les candidats du retrieval), **taux de citations valides** (vérifie que les snippets renvoyés existent dans la page référencée).

```

python

# kaitiaki/eval/evaluate.py
import os, sys, json, time, statistics
from pathlib import Path
from typing import Dict, List, Tuple
import requests
import yaml

# Permet d'importer les modules du projet
ROOT = Path(__file__).resolve().parents[1]
if str(ROOT) not in sys.path:
    sys.path.insert(0, str(ROOT))

from kaitiaki.rag.pipeline import hybrid_search
from kaitiaki.rag.schemas import Answer # seulement pour le format

CFG = yaml.safe_load((ROOT / "config" / "app.yaml").read_text(encoding="utf-8"))

API_URL = os.environ.get("KAI_API", "http://127.0.0.1:8000")
TESTSET_PATH = ROOT / "eval" / "qa_testset.json"
PAGES_DIR = ROOT / "data" / "processed"

def load_testset() -> List[Dict]:
    return json.loads(TESTSET_PATH.read_text(encoding="utf-8"))

def load_page_text(doc_id: str, page: int) -> str:
    """Charge le texte brut d'une page à partir de *.pages.json généré par parse_pdf.py"""
    pages_file = PAGES_DIR / f"{Path(doc_id).stem}.pages.json"
    if not pages_file.exists():
        return ""
    data = json.loads(pages_file.read_text(encoding="utf-8"))
    for p in data.get("pages", []):
        if int(p.get("page", -1)) == int(page):
            return (p.get("text") or "").strip()
    return ""

def call_api(question: str) -> Tuple[Dict, int]:

```

```

"""Appel end-to-end pour mesurer la latence globale ; retourne (json, wall_ms)"""
t0 = time.time()
r = requests.post(f"{API_URL}/query", json={"text": question})
r.raise_for_status()
wall_ms = int((time.time() - t0) * 1000)
return r.json(), wall_ms

def topk_docpage_from_hybrid(question: str, k: int = 20) -> List[Tuple[str, int]]:
    """Utilise le retrieval pour estimer le recall@k (doc_id, page) après fusion + rerank."""
    ranked, _ = hybrid_search(question, rerank_top_k=k)
    ids = []
    for d, _score in ranked[:k]:
        doc_id = d.meta.get("doc_id")
        page = int(d.meta.get("page", -1))
        ids.append((doc_id, page))
    return ids

def citations_validity(resp: Dict, snippet_min_len: int = 40) -> float:
    """Vérifie que chaque citation est plausible:
    - présence d'une page dans le doc référencé
    - le snippet (si présent) existe réellement dans le texte de la page
    """
    cits = resp.get("citations", [])
    if not cits:
        return 0.0
    ok = 0
    for c in cits:
        doc_id = c.get("doc_id")
        page = c.get("page")
        snippet = (c.get("snippet") or "").strip()
        page_text = load_page_text(doc_id, page)
        cond_page = len(page_text) > 0
        cond_snip = (len(snippet) >= snippet_min_len and snippet in page_text) if snippet else
True
        if cond_page and cond_snip:
            ok += 1
    return ok / len(cits)

def recall_at_k(gt: List[Dict], retrieved: List[Tuple[str, int]], k: int = 20) -> float:
    """gt: [{"doc_id":..., "page":...}, ...] ; retrieved: [(doc_id,page), ...]"""
    if not gt:
        return 1.0
    retrieved_k = set(retrieved[:k])
    hits = sum(1 for g in gt if (g["doc_id"], int(g.get("page", -1))) in retrieved_k)
    return hits / len(gt)

def run_eval():
    tests = load_testset()
    latencies = [] # p95 calculé sur latence end-to-end API
    recalls = [] # recall@20 basé sur retrieval
    cit_valid_rates = [] # taux de citations valides (0..1)
    per_item = []

    for i, t in enumerate(tests, start=1):
        q = t["question"]
        gt = t.get("expected_citations", [])
        # 1) retrieval pour recall@20
        retrieved = topk_docpage_from_hybrid(q, k=20)
        r_at_20 = recall_at_k(gt, retrieved, k=20)
        recalls.append(r_at_20)

        # 2) appel API pour latence & citations
        try:
            resp, wall_ms = call_api(q)
        except Exception as e:
            per_item.append({
                "i": i, "question": q, "error": str(e), "recall@20": r_at_20
            })
            latencies.append(99999)
            cit_valid_rates.append(0.0)
            continue

        latencies.append(wall_ms if "latency_ms" not in resp else resp["latency_ms"])
        cit_rate = citations_validity(resp)
        cit_valid_rates.append(cit_rate)

```

```

per_item.append({
    "i": i,
    "question": q,
    "recall@20": round(r_at_20, 3),
    "citations_valid_rate": round(cit_rate, 3),
    "latency_ms": resp.get("latency_ms", wall_ms),
    "n_citations": len(resp.get("citations", []))
})

# Agrégats
p95 = int(statistics.quantiles(latencies, n=100)[94]) if len(latencies) >= 2 else
latencies[0]
report = {
    "n": len(tests),
    "latency_p95_ms": p95,
    "latency_avg_ms": int(sum(latencies) / max(1, len(latencies))),
    "recall@20_avg": round(sum(recalls)/max(1, len(recalls)), 3),
    "cit_valid_rate_avg": round(sum(cit_valid_rates)/max(1, len(cit_valid_rates)), 3),
    "items": per_item
}

out_json = ROOT / "eval" / "report.json"
out_csv = ROOT / "eval" / "report.csv"

out_json.write_text(json.dumps(report, ensure_ascii=False, indent=2), encoding="utf-8")

# CSV simple
lines = ["i,question,recall_at_20,citations_valid_rate,latency_ms,n_citations"]
for it in per_item:
    lines.append(f'{it.get("i")},{it.get("question").replace(",",";")},
{it.get("recall@20",0)},{it.get("citations_valid_rate",0)},{it.get("latency_ms",0)},
{it.get("n_citations",0)}')
out_csv.write_text("\n".join(lines), encoding="utf-8")

print("=== Résumé ===")
print(json.dumps({k:v for k,v in report.items() if k != "items"}, ensure_ascii=False,
indent=2))
print(f"\nDétails enregistrés dans: {out_json}\nTableau: {out_csv}")

if __name__ == "__main__":
    run_eval()

```

Utilisation

```

bash

# 1) Démarrer votre serveur API local:
uvicorn kaitiaki.api.server:app --reload

# 2) Lancer l'évaluation (dans un autre terminal):
python -m kaitiaki.eval.evaluate

```

2) Interprétation des métriques

- **latency_p95_ms** : 95 % des requêtes sont servies sous ce temps (end-to-end /query).
- **recall@20_avg** : proportion moyenne des références attendues (doc_id,page) présentes dans les 20 premiers candidats du retrieval.
- **cit_valid_rate_avg** : part des citations dont le **snippet** apparaît bien dans le texte de la **page** référencée.

3) Checklist de validation V1 (à cocher)

Données et ingestion

- ☐ Les PDF test (JONC/IEOM/ISEE/délibérations) sont présents dans `data/raw/`.
- ☐ `parse_pdf.py` a généré les `*.pages.json` dans `data/processed/`.
- ☐ `normalize.py` a produit les `*.normalized.json` (chunks).
- ☐ `indexer.py` a créé la collection Qdrant, les embeddings et l'index BM25 local (fichiers `bm25_index.pkl` et `bm25_meta.json`).

Fonctionnel

- ☐ L'endpoint `/health` répond `{ok:true}`.
- ☐ L'endpoint `/query` renvoie une **réponse** + au moins **2 citations** avec `doc_id` et `page`.
- ☐ Les citations ouvrent (ou correspondent à) des pages existantes dans `*.pages.json`.

Qualité (seuils de sortie V1 — à ajuster selon matériel)

- ☐ `latency_p95_ms` ≤ 5000 ms (CPU quantisé) ou ≤ 2500 ms (GPU 7-8B).
- ☐ `recall@20_avg` $\geq 0,70$ sur le testset interne (après tuning `top_k` / chunking).
- ☐ `cit_valid_rate_avg` $\geq 0,80$ (snippets réellement présents dans la page).
- ☐ **Erreurs API** (timeouts/500) < 2 % sur 50 requêtes.

Observabilité / Logs

- ☐ Les temps de réponse et tailles de contextes sont tracés (latence totale, tokens LLM).
- ☐ Un journal de requêtes est conservé localement (date/heure, question, `doc_ids` cités).

Sécurité / Conformité

- ☐ Mention visible : « Kaitiaki fournit des synthèses d'informations publiques. Ce n'est pas un avis juridique. »
- ☐ Les chemins d'accès aux documents ne quittent pas l'infrastructure locale.
- ☐ Les clés/modèles (si utilisés) ne sont pas committés (gitignored).

Conseils de tuning rapide si seuils non atteints

- Latence**
 - Réduire `rerank_top_k` à 15-20.
 - Quantiser le LLM (GGUF) et/ou baisser `max_tokens`.
 - Mettre en cache les embeddings de requêtes chaudes.

- **Recall@20**

- Augmenter **top_k_dense** et/ou **top_k_bm25** (ex. 40/40) puis fusion RRF.
- Améliorer le **chunking** (chevauchement 200–300, taille 1200–1600).
- Enrichir métadonnées (source, date) et appliquer des filtres temporels avant fusion.

- **Citations valides**

- Générer les citations à partir des **passages réellement passés au LLM**.
- Augmenter la longueur minimale des snippets gardés (≥ 60 –80 caractères).
- Vérifier l'encodage des PDF (OCR si pages scannées).

Vous pouvez copier ces deux fichiers, ajuster 2–3 chemins dans **app.yaml**, remplir **qa_testset.json** avec vos documents, et lancer **python -m kaitiaki.eval.evaluate**.

Souhaitez-vous que j'ajoute un **script shell** d'exécution bout-à-bout (ingestion → serveur → évaluation) pour automatiser vos tests locaux ?



Crée tous les fichiers de documentation de dev, install, conception, test, etc. Format md.
Répertoire documentation, que tu organiseras de manière professionnelle

Voici un dossier **documentation/** structuré de façon professionnelle, avec des fichiers Markdown prêts à déposer dans votre dépôt. Chaque fichier contient un contenu utile dès maintenant et des emplacements clairs pour compléter au fil du projet.

```
pgsql

documentation/
  README.md
  00-produit/
    overview.md
    scope.md
    disclaimer.md
  01-architecture/
    architecture.md
    rag-pipeline.md
    data-model.md
    security.md
    operations.md
  02-installation/
    prerequisites.md
    quickstart.md
    configuration.md
    run-local.md
  03-developpement/
    repo-structure.md
    coding-standards.md
    contributing.md
    api-spec.md
    data-ingestion.md
    tuning-retrieval.md
  04-tests/
    test-plan.md
    evaluation-metrics.md
    manual-checklist.md
  05-operations/
    monitoring.md
```

```

backup-restore.md
troubleshooting.md
06-legal/
  licenses.md
07-roadmap/
  roadmap.md
  changelog.md
08-branding/
  naming.md
  ui-copy.md

```

documentation/README.md

```

md

# Kaitiaki – Documentation

Cette documentation couvre la conception, l'installation, le développement, les tests,
l'exploitation et la conformité du projet Kaitiaki.

## Plan
- 00-produit/
  - overview.md : présentation du produit
  - scope.md : périmètre fonctionnel et non-fonctionnel
  - disclaimer.md : avertissements et limites d'usage
- 01-architecture/
  - architecture.md : vue d'ensemble technique
  - rag-pipeline.md : pipeline RAG hybride détaillé
  - data-model.md : schémas de données et formats d'échange
  - security.md : sécurité, souveraineté, RBAC
  - operations.md : vues déploiement et exécution
- 02-installation/
  - prerequisites.md : prérequis matériels et logiciels
  - quickstart.md : installation accélérée
  - configuration.md : paramètres applicatifs
  - run-local.md : lancement local sans Docker
- 03-developpement/
  - repo-structure.md : arborescence du repo
  - coding-standards.md : conventions de code
  - contributing.md : workflow Git, PR, releases
  - api-spec.md : contrat d'API FastAPI
  - data-ingestion.md : ingestion, parsing, normalisation, indexation
  - tuning-retrieval.md : réglages dense/BM25/rerank
- 04-tests/
  - test-plan.md : stratégie de test
  - evaluation-metrics.md : métriques (p95, recall@20, citations valides)
  - manual-checklist.md : recette fonctionnelle
- 05-operations/
  - monitoring.md : observabilité
  - backup-restore.md : sauvegarde et restauration
  - troubleshooting.md : résolution d'incidents
- 06-legal/
  - licenses.md : licences logicielles
- 07-roadmap/
  - roadmap.md : phases du produit
  - changelog.md : journal des changements
- 08-branding/
  - naming.md : nommage et positionnement
  - ui-copy.md : ton et messages UI

Pour démarrer, lisez 02-installation/quickstart.md, puis 03-developpement/api-spec.md et
01-architecture/rag-pipeline.md.

```

documentation/00-produit/overview.md

```
md

# Présentation produit

**Kaitiaki** est un assistant de veille stratégique pour la Nouvelle-Calédonie. Il interroge et synthétise des publications publiques (JONC, ISEE, IEOM, délibérations) et fournit des réponses sourcées.

## Objectifs
- Recherche en langage naturel avec réponses concises et citations exactes.
- Synthèse thématique par secteurs (BTP, énergie, emploi).
- Filtrage temporel simple (mois/trimestre/année).
- Évolutivité vers opendata et indicateurs.

## Publics
- Entreprises industrielles, consultants, fédérations professionnelles.
- Journalistes, institutions, directions support.

## Différenciants
- Souveraineté et on-prem par conception.
- Hybridation dense + BM25 + reranking cross-encoder.
- Citations robustes (doc/page/snippet), traçabilité des versions.
```

documentation/00-produit/scope.md

```
md

# Périmètre V1

## Fonctionnel
- Recherche NL sur 6–10 PDF récents (JONC/IEOM/ISEE/délibérations).
- Réponse synthétique avec 2–4 citations cliquables.
- Filtres rudimentaires par source et période (à étendre en V1.1).
- Évaluation de base (latence p95, recall@20, citations valides).

## Hors-périmètre V1
- Pas de conseils juridiques.
- Pas d'authentification fine (SSO/RBAC) en MVP.
- Pas de conteneurisation (Docker) pour la V1.

## Non-fonctionnel
- Déploiement local sans Docker.
- Temps de réponse p95 visé: ≤ 5 s CPU quantisé, ≤ 2.5 s GPU 7–8B.
- Journalisation locale des requêtes et des versions indexées.
```

documentation/00-produit/disclaimer.md

```
md

# Avertissements

- Kaitiaki fournit des **synthèses d'informations publiques**. Il ne s'agit **pas** d'un avis juridique ou d'un conseil professionnel.
- Les réponses doivent être **vérifiées** via les citations et documents sources.
- En cas d'ambiguïté, la source officielle prévaut.
```

documentation/01-architecture/architecture.md

md

Architecture technique

Vue d'ensemble

- Front minimal : FastAPI + templates HTML.
- Backend : Python 3.11, FastAPI.
- Orchestration RAG : Haystack 2.x.
- Stockage :
 - Qdrant pour embeddings denses.
 - BM25 local (rank-bm25, pickle) en MVP.
- Modèles :
 - Embeddings : all-MiniLM-L6-v2 (ou équivalent FR/EN).
 - Reranker : BAAI/bge-reranker-base.
 - LLM : Llama 3.1-8B ou Mistral 7B via API locale OpenAI-like.

Flux

1. Ingestion PDF → extraction texte → normalisation → chunking.
2. Indexation :
 - Écriture des chunks dans Qdrant + embeddings.
 - Construction d'un index BM25 local.
3. Requête :
 - Dense top-K + BM25 top-K → fusion RRF → reranking cross-encoder → génération LLM.
4. Restitution : réponse + citations (doc/page/snippet).

Évolutions

- Ajout OpenSearch pour BM25 avancé, filtres par champs et agrégations.
- Ajout opendata pour indicateurs (tableaux/graphes).
- Couches KG/ontologie ciblées (post-V1).

documentation/01-architecture/rag-pipeline.md

md

Pipeline RAG hybride

Étapes

1. Prétraitement : extraction, nettoyage, segmentation (1 200–1 600 caractères, overlap 200–300).
2. Retrieval :
 - Dense (embeddings) top_k_dense par similarité cosinus.
 - BM25 top_k_bm25 via index local.
3. Fusion : Reciprocal Rank Fusion (param k=60 par défaut).
4. Reranking : cross-encoder sur top-K fusionné (25 par défaut).
5. Génération : prompt "quote-first", température 0.2, max_tokens 600.

Paramètres par défaut

- top_k_dense: 20
- top_k_bm25: 20
- rerank_top_k: 25
- fusion: RRF (k=60)

Citations

- doc_id, page, extrait (≥ 60 caractères).
- Conformés au contenu réellement passé au LLM.

documentation/01-architecture/data-model.md

md

Modèle de données

Fichiers intermédiaires

```
- `*.pages.json` :
- doc_id, pages[{page:int, text:str}]
- `*.normalized.json` :
- doc_id, date, chunks[{doc_id, date, page, text, source}]

## Indices
- Qdrant : documents = chunks, metadata = {doc_id, page, date, source}, embedding 384-d.
- BM25 local : tokenisation simple FR/EN, fichiers `bm25_index.pkl`, `bm25_meta.json`.

## API
- POST /query : {text, sources?, date_from?, date_to?, top_k?}
- Réponse : {answer, citations[], latency_ms}
```

documentation/01-architecture/security.md

md

Sécurité et souveraineté

- Données locales uniquement, pas d'appel cloud obligatoire.
- Chiffrement disque conseillé (LUKS/BitLocker) pour environnements sensibles.
- Journalisation locale des requêtes (sans PII).
- Clés de services ou endpoints LLM locaux hors repo (fichier .env, non versionné).
- Avertissement utilisateur visible : pas d'avis juridique.

documentation/01-architecture/operations.md

md

Opérations

Services

- Qdrant local, port 6333.
- Serveur LLM local OpenAI-like (vLLM, llama.cpp wrapper), port configurable.
- FastAPI (uvicorn) pour l'API et l'UI.

Démarrage

1. Préparer données dans `kaitiaki/data/raw/`.
2. Ingestion : parse → normalize → indexer.
3. Lancer API : `uvicorn kaitiaki.api.server:app --reload`.

Journalisation

- Logs applicatifs standard.
- `eval/report.json` et `eval/report.csv` pour les campagnes de tests.

documentation/02-installation/prerequisites.md

md

Prérequis

- OS : Linux/macOS/Windows.
- Python 3.11+.
- Qdrant binaire démarré localement (port 6333).
- Modèle LLM local accessible via API OpenAI-compatible.
- Si CPU uniquement : utiliser un modèle quantisé (GGUF) et limiter max_tokens.

documentation/02-installation/quickstart.md

md

Démarrage rapide

```
```bash
python -m venv .venv && source .venv/bin/activate
pip install -r requirements.txt

Déposer 6-10 PDF dans kaitiaki/data/raw/
python -m kaitiaki.ingest.parse_pdf
python -m kaitiaki.ingest.normalize
python -m kaitiaki.ingest.indexer

uvicorn kaitiaki.api.server:app --reload
Ouvrir http://127.0.0.1:8000
```

Pour l'évaluation :

bash

```
python -m kaitiaki.eval.evaluate
```

yaml

```

documentation/02-installation/configuration.md

```md
# Configuration

Fichier : `kaitiaki/config/app.yaml`

- paths : chemins des données et index BM25.
- qdrant : host, port, collection.
- models :
  - embedding
  - reranker
  - llm_base_url
  - llm_model
- retrieval :
  - top_k_dense, top_k_bm25, fusion, rerank_top_k
```

documentation/02-installation/run-local.md

md

Exécution locale sans Docker

1. Démarrer Qdrant : binaire ou service.
2. Démarrer le serveur LLM local (OpenAI-like).
3. Créer l'environnement Python, installer requirements.
4. Ingestion des PDF.
5. Lancer FastAPI.

Astuce : pour un poste sans GPU, réduire `rerank_top_k` à 15-20 et `max_tokens` à 400.

documentation/03-developpement/repo-structure.md

md

Arborescence du dépôt

- ``kaitiaki/ingest/`` : *parse_pdf, normalize, indexer.*
- ``kaitiaki/rag/`` : *pipeline hybride, fusion RRF, client LLM, schémas Pydantic.*
- ``kaitiaki/api/`` : *serveur FastAPI, templates HTML.*
- ``kaitiaki/eval/`` : *tests et métriques.*
- ``kaitiaki/utils/`` : *logging, helpers.*
- ``documentation/`` : *ce dossier.*

documentation/03-developpement/coding-standards.md

md

Conventions de code

- Python 3.11+, typage statique (typing), docstrings Google style.
- Formatage : black, isort, flake8 (optionnel).
- Nommage : *snake_case* pour fonctions/variables, *PascalCase* pour classes.
- Pas de secrets en clair ; variables d'environnement ou fichiers ignorés par Git.

documentation/03-developpement/contributing.md

md

Contribuer

Workflow

- branche ``main`` protégée.
- branches de fonctionnalité : ``feat/...``, corrections : ``fix/...``.
- PR avec description, tests, mise à jour de la doc si nécessaire.
- Revue obligatoire sur PRs significatives.

Commits

- message clair au présent.
- référence au ticket interne si applicable.

Releases

- tag sémantique (v0.1.0).
- mise à jour de 07-roadmap/changelog.md.

documentation/03-developpement/api-spec.md

md

Spécification API

POST /query

Entrée :

````json``

```
{
 "text": "Quelles mesures récentes concernant le BTP ?",
 "sources": ["JONC", "IEOM", "ISEE"],
 "date_from": "2025-05-01",
```

```
"date_to": "2025-08-25",
"top_k": 20
}
```

Sortie :

```
json

{
 "answer": "Synthèse ...",
 "citations": [
 {"doc_id": "jonc_2025-06-12.pdf", "page": 4, "snippet": "..."}
],
 "latency_ms": 820
}
```

## POST /ingest

- Réindexe les fichiers présents dans **data/raw/**.
- Réponse : { "status": "ok" } ou détails.

## GET /health

- Statut simple { "ok": true }.

```
yaml

documentation/03-developpement/data-ingestion.md

```md
# Ingestion et indexation

1. `parse_pdf.py` : PDF → `*.pages.json` (page, texte).
2. `normalize.py` : normalise, segmente en chunks → `*.normalized.json`.
3. `indexer.py` :
   - Écrit les chunks dans Qdrant.
   - Encode embeddings denses et met à jour Qdrant.
   - Construit l'index BM25 local (rank-bm25).

Conseils :
- Taille chunk 1 200–1 600, overlap 200–300.
- Ajouter `source`, `date`, `doc_id`, `page` dans metadata.
```

documentation/03-developpement/tuning-retrieval.md

```
md

# Réglage retrieval

- Augmenter `top_k_dense` et `top_k_bm25` à 40/40 si recall faible.
- Réduire `rerank_top_k` si latence élevée.
- Améliorer le chunking et la qualité de l'OCR.
- Activer OpenSearch en V1.1 pour filtres lexicaux avancés.
```


documentation/04-tests/test-plan.md

```
md

# Plan de tests

## Tests automatisés
- `eval/evaluate.py` :
  - latence p95 end-to-end,
  - recall@20,
  - taux de citations valides.

## Tests manuels
- Recette UI : saisie requêtes, vérification citations, navigation.
- Robustesse : 50 requêtes variées, mesure erreurs/temps.

## Jeux d'essai
- `eval/qa_testset.json` : 10–15 Q/A couvrant JONC, IEOM, ISEE, délibérations.
```

documentation/04-tests/evaluation-metrics.md

```
md

# Métriques d'évaluation

- Latence p95 : 95 % des requêtes sous ce temps.
- Recall@20 : part des références attendues (doc_id, page) présentes dans les 20 premiers candidats.
- Taux de citations valides : proportion des citations dont le snippet apparaît sur la page référencée.

## Seuils V1
- Latence p95  $\leq 5\,000$  ms (CPU) ou  $\leq 2\,500$  ms (GPU).
- Recall@20  $\geq 0,70$ .
- Citations valides  $\geq 0,80$ .
```

documentation/04-tests/manual-checklist.md

```
md

# Checklist de recette fonctionnelle

- Saisie d'une question et obtention d'une réponse en français.
- Présence de 2–4 citations avec doc_id, page, extrait.
- Clic citation → ouverture du document à la page (ou vérification manuelle du texte).
- Filtre source (JONC/IEOM/ISEE/délibérations) fonctionnel.
- Stabilité visuelle et absence d'erreurs console.
```

documentation/05-operations/monitoring.md

```
md

# Observabilité

- Logs applicatifs FastAPI (requêtes, erreurs).
- Journaux de latence et tailles de contextes.
```

- Exports `eval/report.json` et `eval/report.csv` pour campagnes de test.
- Intégration Prometheus/Grafana possible en V1.1.

documentation/05-operations/backup-restore.md

```
md

# Sauvegarde et restauration

## À sauvegarder
- Qdrant : dossier de stockage (collection Kaitiaki).
- BM25 local : `bm25_index.pkl` et `bm25_meta.json`.
- Données sources : `data/raw/` et `data/processed/`.

## Restauration
1. Restaurer le dossier Qdrant.
2. Remettre les fichiers BM25.
3. Rejouer l'ingestion si nécessaire.
```

documentation/05-operations/troubleshooting.md

```
md

# Dépannage

## Problèmes de performance
- Réduire `rerank_top_k` et `max_tokens`.
- Vérifier que le LLM local répond dans les délais.

## Résultats médiocres
- Augmenter top_k_dense/BM25.
- Ajuster chunking et overlap.
- Qualité OCR insuffisante.

## Erreurs de citations
- Vérifier que les snippets proviennent des passages réellement fournis au LLM.
- Augmenter la longueur minimale des snippets.
```

documentation/06-legal/licenses.md

```
md

# Licences

- Bibliothèques Python : se référer au fichier `requirements.txt`.
- Modèles : vérifier les licences des modèles utilisés (embeddings, reranker, LLM).
- Données : les publications publiques restent la propriété des émetteurs ; respecter les conditions d'utilisation.
```

documentation/07-roadmap/roadmap.md

md

Roadmap

V1 (MVP)

- RAG hybride local (Qdrant + BM25 local).
- UI minimaliste, citations cliquables.
- Évaluation basique et seuils de sortie.

V1.1

- OpenSearch pour BM25 avancé et filtres par champs.
- Alertes hebdomadaires par email.

V2

- Intégration opendata (indicateurs avec graphiques).
- Filtres thématiques et taxonomie.
- Authentification basique.

V3

- Graphe de connaissances ciblé.
- Ontologie restreinte pour explicabilité.

documentation/07-roadmap/changelog.md

md

Changelog

[Unreleased]

- Initialisation de la documentation.

v0.1.0 - MVP

- Ingestion PDF, indexation Qdrant + BM25 local.
- Pipeline RAG hybride + reranking.
- API FastAPI et UI simple.
- Évaluation et checklist V1.

documentation/08-branding/naming.md

md

Naming

- Nom produit : **Kaitiaki**
- Sens : gardien, protecteur, veilleur.
- Variantes dépôt :
 - `kaitiaki`
 - `kaitiaki-nc`

documentation/08-branding/ui-copy.md

md

UI Copy

- Titre : Kaitiaki – Veille NC
- Champ de recherche : "Saisissez votre question"
- Bouton : "Chercher"

- Avertissement : "Kaitiaki fournit des synthèses d'informations publiques. Ce n'est pas un avis juridique."
 - Messages d'erreur : "Aucune source trouvée pour cette requête", "Temps de réponse dépassé"

Souhaitez-vous que je vous fournisse aussi un script de création automatique de cette arborescence et des fichiers (un script Python qui écrit ces `.md` dans `documentation/`), afin que vous puissiez générer le dossier d'un seul coup sur votre machine ?



oui

Voici un script unique qui crée toute l'arborescence `documentation/` et y dépose les fichiers Markdown avec les contenus proposés. Il n'écrase pas les fichiers existants par défaut (option `--force` pour forcer l'écrasement).

Enregistrez ce script, par exemple sous `scripts/bootstrap_docs.py`, puis exécutez-le depuis la racine de votre dépôt.

bash

```
python3 scripts/bootstrap_docs.py
# ou pour écraser les fichiers existants
python3 scripts/bootstrap_docs.py --force
```

python

```
#!/usr/bin/env python3
# scripts/bootstrap_docs.py
from __future__ import annotations
import argparse
from pathlib import Path

ROOT = Path(__file__).resolve().parents[1]
DOC = ROOT / "documentation"

FILES: dict[str, str] = {
    # ===== documentation/ =====
    "documentation/README.md": """# Kaitiaki – Documentation
```

Cette documentation couvre la conception, l'installation, le développement, les tests, l'exploitation et la conformité du projet **Kaitiaki**.

```
## Plan
- 00-produit/
  - overview.md : présentation du produit
  - scope.md : périmètre fonctionnel et non-fonctionnel
  - disclaimer.md : avertissements et limites d'usage
- 01-architecture/
  - architecture.md : vue d'ensemble technique
  - rag-pipeline.md : pipeline RAG hybride détaillé
  - data-model.md : schémas de données et formats d'échange
  - security.md : sécurité, souveraineté, RBAC
  - operations.md : vues déploiement et exécution
- 02-installation/
  - prerequisites.md : prérequis matériels et logiciels
  - quickstart.md : installation accélérée
  - configuration.md : paramètres applicatifs
  - run-local.md : lancement local sans Docker
- 03-developpement/
  - repo-structure.md : arborescence du repo
```

- coding-standards.md : conventions de code
- contributing.md : workflow Git, PR, releases
- api-spec.md : contrat d'API FastAPI
- data-ingestion.md : ingestion, parsing, normalisation, indexation
- tuning-retrieval.md : réglages dense/BM25/rerank
- 04-tests/
 - test-plan.md : stratégie de test
 - evaluation-metrics.md : métriques (p95, recall@20, citations valides)
 - manual-checklist.md : recette fonctionnelle
- 05-operations/
 - monitoring.md : observabilité
 - backup-restore.md : sauvegarde et restauration
 - troubleshooting.md : résolution d'incidents
- 06-legal/
 - licenses.md : licences logicielles
- 07-roadmap/
 - roadmap.md : phases du produit
 - changelog.md : journal des changements
- 08-branding/
 - naming.md : nommage et positionnement
 - ui-copy.md : ton et messages UI

Pour démarrer, lisez ****02-installation/quickstart.md****, puis ****03-developpement/api-spec.md**** et ****01-architecture/rag-pipeline.md****.

```

"""
# ===== 00-produit =====
"documentation/00-produit/overview.md": """# Présentation produit

**Kaitiaki** est un assistant de veille stratégique pour la Nouvelle-Calédonie. Il interroge et synthétise des publications publiques (JONC, ISEE, IEOM, délibérations) et fournit des réponses sourcées.

## Objectifs
- Recherche en langage naturel avec réponses concises et citations exactes.
- Synthèse thématique par secteurs (BTP, énergie, emploi).
- Filtrage temporel simple (mois/trimestre/année).
- Évolutivité vers opendata et indicateurs.

## Publics
- Entreprises industrielles, consultants, fédérations professionnelles.
- Journalistes, institutions, directions support.

## Différenciants
- Souveraineté et on-prem par conception.
- Hybridation dense + BM25 + reranking cross-encoder.
- Citations robustes (doc/page/snippet), traçabilité des versions.
"""

"documentation/00-produit/scope.md": """# Périmètre V1

## Fonctionnel
- Recherche NL sur 6–10 PDF récents (JONC/IEOM/ISEE/délibérations).
- Réponse synthétique avec 2–4 citations cliquables.
- Filtres rudimentaires par source et période (à étendre en V1.1).
- Évaluation de base (latence p95, recall@20, citations valides).

## Hors-périmètre V1
- Pas de conseils juridiques.
- Pas d'authentification fine (SSO/RBAC) en MVP.
- Pas de conteneurisation (Docker) pour la V1.

## Non-fonctionnel
- Déploiement local sans Docker.
- Temps de réponse p95 visé: ≤ 5 s CPU quantisé, ≤ 2.5 s GPU 7–8B.
- Journalisation locale des requêtes et des versions indexées.
"""

"documentation/00-produit/disclaimer.md": """# Avertissements

- Kaitiaki fournit des **synthèses d'informations publiques**. Il ne s'agit **pas** d'un avis juridique ou d'un conseil professionnel.
- Les réponses doivent être **vérifiées** via les citations et documents sources.
- En cas d'ambiguïté, la source officielle prévaut.
"""

```

```
# ===== 01-architecture =====
"documentation/01-architecture/architecture.md": """# Architecture technique

## Vue d'ensemble
- Front minimal : FastAPI + templates HTML.
- Backend : Python 3.11, FastAPI.
- Orchestration RAG : Haystack 2.x.
- Stockage :
  - Qdrant pour embeddings denses.
  - BM25 local (rank-bm25, pickle) en MVP.
- Modèles :
  - Embeddings : all-MiniLM-L6-v2 (ou équivalent FR/EN).
  - Reranker : BAAI/bge-reranker-base.
  - LLM : Llama 3.1-8B ou Mistral 7B via API locale OpenAI-like.

## Flux
1. Ingestion PDF → extraction texte → normalisation → chunking.
2. Indexation :
  - Écriture des chunks dans Qdrant + embeddings.
  - Construction d'un index BM25 local.
3. Requête :
  - Dense top-K + BM25 top-K → fusion RRF → reranking cross-encoder → génération LLM.
4. Restitution : réponse + citations (doc/page/snippet).

## Évolutions
- Ajout OpenSearch pour BM25 avancé, filtres par champs et agrégations.
- Ajout opendata pour indicateurs (tableaux/graphes).
- Couches KG/ontologie ciblées (post-V1).
""",

"documentation/01-architecture/rag-pipeline.md": """# Pipeline RAG hybride

## Étapes
1. Prétraitement : extraction, nettoyage, segmentation (1 200–1 600 caractères, overlap 200–300).
2. Retrieval :
  - Dense (embeddings) top_k_dense par similarité cosinus.
  - BM25 top_k_bm25 via index local.
3. Fusion : Reciprocal Rank Fusion (param k=60 par défaut).
4. Reranking : cross-encoder sur top-K fusionné (25 par défaut).
5. Génération : prompt "quote-first", température 0.2, max_tokens 600.

## Paramètres par défaut
- top_k_dense: 20
- top_k_bm25: 20
- rerank_top_k: 25
- fusion: RRF (k=60)

## Citations
- doc_id, page, extrait (≥ 60 caractères).
- Conformés au contenu réellement passé au LLM.
""",

"documentation/01-architecture/data-model.md": """# Modèle de données

## Fichiers intermédiaires
- `*.pages.json` :
  - doc_id, pages[{page:int, text:str}]
- `*.normalized.json` :
  - doc_id, date, chunks[{doc_id, date, page, text, source}]

## Indices
- Qdrant : documents = chunks, metadata = {doc_id, page, date, source}, embedding 384-d.
- BM25 local : tokenisation simple FR/EN, fichiers `bm25_index.pkl`, `bm25_meta.json`.

## API
- POST /query : {text, sources?, date_from?, date_to?, top_k?}
- Réponse : {answer, citations[], latency_ms}
""",

"documentation/01-architecture/security.md": """# Sécurité et souveraineté

- Données locales uniquement, pas d'appel cloud obligatoire.
- Chiffrement disque conseillé (LUKS/BitLocker) pour environnements sensibles.
```

```

- Journalisation locale des requêtes (sans PII).
- Clés de services ou endpoints LLM locaux hors repo (fichier .env, non versionné).
- Avertissement utilisateur visible : pas d'avis juridique.
"""

"documentation/01-architecture/operations.md": """# Opérations

## Services
- Qdrant local, port 6333.
- Serveur LLM local OpenAI-like (vLLM, llama.cpp wrapper), port configurable.
- FastAPI (uvicorn) pour l'API et l'UI.

## Démarrage
1. Préparer données dans `kaitiaki/data/raw/`.
2. Ingestion : parse → normalize → indexer.
3. Lancer API : `uvicorn kaitiaki.api.server:app --reload`.

## Journalisation
- Logs applicatifs standard.
- `eval/report.json` et `eval/report.csv` pour les campagnes de tests.
"""

# ===== 02-installation =====
"documentation/02-installation/prerequisites.md": """# Prérequis

- OS : Linux/macOS/Windows.
- Python 3.11+.
- Qdrant binaire démarré localement (port 6333).
- Modèle LLM local accessible via API OpenAI-compatible.
- Si CPU uniquement : utiliser un modèle quantisé (GGUF) et limiter max_tokens.
"""

"documentation/02-installation/quickstart.md": """# Démarrage rapide

```bash
python -m venv .venv && source .venv/bin/activate
pip install -r requirements.txt

Déposer 6–10 PDF dans kaitiaki/data/raw/
python -m kaitiaki.ingest.parse_pdf
python -m kaitiaki.ingest.normalize
python -m kaitiaki.ingest.indexer

uvicorn kaitiaki.api.server:app --reload
Ouvrir http://127.0.0.1:8000

```

Pour l'évaluation :

```

bash

python -m kaitiaki.eval.evaluate

```

"""

```

bash

"documentation/02-installation/configuration.md": """# Configuration

```

Fichier : **kaitiaki/config/app.yaml**

- paths : chemins des données et index BM25.
- qdrant : host, port, collection.
- models :
  - embedding

- reranker
  - llm\_base\_url
  - llm\_model
  - retrieval :
    - top\_k\_dense, top\_k\_bm25, fusion, rerank\_top\_k
- ```
"""
"documentation/02-installation/run-local.md": """# Exécution locale sans Docker
```

1. Démarrer Qdrant : binaire ou service.
2. Démarrer le serveur LLM local (OpenAI-like).
3. Créer l'environnement Python, installer requirements.
4. Ingestion des PDF.
5. Lancer FastAPI.

Astuce : pour un poste sans GPU, réduire **rerank_top_k** à 15–20 et **max_tokens** à 400.

```
"""
bash

# ===== 03-developpement =====
"documentation/03-developpement/repo-structure.md": """# Arborescence du dépôt
```

- **kaitiaki/ingest/** : parse_pdf, normalize, indexer.
 - **kaitiaki/rag/** : pipeline hybride, fusion RRF, client LLM, schémas Pydantic.
 - **kaitiaki/api/** : serveur FastAPI, templates HTML.
 - **kaitiaki/eval/** : tests et métriques.
 - **kaitiaki/utils/** : logging, helpers.
 - **documentation/** : ce dossier.
- ```
"""
"documentation/03-developpement/coding-standards.md": """# Conventions de code
```
- Python 3.11+, typage statique (typing), docstrings Google style.
  - Formatage : black, isort, flake8 (optionnel).
  - Nommage : snake\_case pour fonctions/variables, PascalCase pour classes.
  - Pas de secrets en clair ; variables d'environnement ou fichiers ignorés par Git.
- ```
"""
"documentation/03-developpement/contributing.md": """# Contribuer
```

Workflow

- branche **main** protégée.
- branches de fonctionnalité : **feat/...**, corrections : **fix/...**

- PR avec description, tests, mise à jour de la doc si nécessaire.
- Revue obligatoire sur PRs significatives.

Commits

- message clair au présent.
- référence au ticket interne si applicable.

Releases

- tag sémantique (v0.1.0).
- mise à jour de 07-roadmap/changelog.md.

```

"""
"documentation/03-developpement/api-spec.md": """# Spécification API

```

POST /query

Entrée :

```

json

{
  "text": "Quelles mesures récentes concernent le BTP ?",
  "sources": ["JONC", "IEOM", "ISEE"],
  "date_from": "2025-05-01",
  "date_to": "2025-08-25",
  "top_k": 20
}

```

Sortie :

```

json

{
  "answer": "Synthèse ...",
  "citations": [
    {"doc_id": "jonc_2025-06-12.pdf", "page": 4, "snippet": "..."}
  ],
  "latency_ms": 820
}

```

POST /ingest

- Réindexe les fichiers présents dans **data/raw/**.
- Réponse : { "status": "ok" } ou détails.

GET /health

- Statut simple { "ok": true }.
- ```

"""

```

"documentation/03-developpement/data-ingestion.md": ""# Ingestion et indexation

1. **parse\_pdf.py** : PDF → \*.**pages.json** (page, texte).
2. **normalize.py** : normalise, segmente en chunks → \*.**normalized.json**.
3. **indexer.py** :
  - Écrit les chunks dans Qdrant.
  - Encode embeddings denses et met à jour Qdrant.
  - Construit l'index BM25 local (rank-bm25).

Conseils :

- Taille chunk 1 200–1 600, overlap 200–300.
- Ajouter **source, date, doc\_id, page** dans metadata.  
 """,  
 "documentation/03-developpement/tuning-retrieval.md": ""# Réglage retrieval
- Augmenter **top\_k\_dense** et **top\_k\_bm25** à 40/40 si recall faible.
- Réduire **rerank\_top\_k** si latence élevée.
- Améliorer le chunking et la qualité de l'OCR.
- Activer OpenSearch en V1.1 pour filtres lexicaux avancés.  
 """,

## ===== 04-tests =====

"documentation/04-tests/test-plan.md": ""# Plan de tests

## Tests automatisés

- **eval/evaluate.py** :
  - latence p95 end-to-end,
  - recall@20,
  - taux de citations valides.

## Tests manuels

- Recette UI : saisie requêtes, vérification citations, navigation.
- Robustesse : 50 requêtes variées, mesure erreurs/temps.

## Jeux d'essai

- **eval/qa\_testset.json** : 10–15 Q/A couvrant JONC, IEOM, ISEE, délibérations.  
 """,  
 "documentation/04-tests/evaluation-metrics.md": ""# Métriques d'évaluation

- Latence p95 : 95 % des requêtes sous ce temps.
- Recall@20 : part des références attendues (doc\_id,page) présentes dans les 20 premiers candidats.
- Taux de citations valides : proportion des citations dont le snippet apparaît sur la page référencée.

## Seuils V1

- Latence p95  $\leq 5\,000$  ms (CPU) ou  $\leq 2\,500$  ms (GPU).
- Recall@20  $\geq 0,70$ .
- Citations valides  $\geq 0,80$ .
- Saisie d'une question et obtention d'une réponse en français.
- Présence de 2-4 citations avec doc\_id, page, extrait.
- Clic citation → ouverture du document à la page (ou vérification manuelle du texte).
- Filtre source (JONC/IEOM/ISEE/délibérations) fonctionnel.
- Stabilité visuelle et absence d'erreurs console.

## ===== 05-operations =====

- Logs applicatifs FastAPI (requêtes, erreurs).
- Journaux de latence et tailles de contextes.
- Exports `eval/report.json` et `eval/report.csv` pour campagnes de test.
- Intégration Prometheus/Grafana possible en V1.1.
- Sauvegarde et restauration

## À sauvegarder

- Qdrant : dossier de stockage (collection Kaitiaki).
- BM25 local : `bm25_index.pkl` et `bm25_meta.json`.
- Données sources : `data/raw/` et `data/processed/`.

## Restauration

1. Restaurer le dossier Qdrant.
2. Remettre les fichiers BM25.
3. Rejouer l'ingestion si nécessaire.

"documentation/05-operations/troubleshooting.md": ""# Dépannage

## Problèmes de performance

- Réduire `rerank_top_k` et `max_tokens`.
- Vérifier que le LLM local répond dans les délais.

## Résultats médiocres

- Augmenter `top_k_dense/BM25`.
- Ajuster chunking et overlap.
- Qualité OCR insuffisante.

## Erreurs de citations

- Vérifier que les snippets proviennent des passages réellement fournis au LLM.
- Augmenter la longueur minimale des snippets.

""",

### ===== 06-legal =====

"documentation/06-legal/licenses.md": ""# Licences

- Bibliothèques Python : se référer au fichier `requirements.txt`.
- Modèles : vérifier les licences des modèles utilisés (embeddings, reranker, LLM).
- Données : les publications publiques restent la propriété des émetteurs ; respecter les conditions d'utilisation.

""",

### ===== 07-roadmap =====

"documentation/07-roadmap/roadmap.md": ""# Roadmap

## V1 (MVP)

- RAG hybride local (Qdrant + BM25 local).
- UI minimaliste, citations cliquables.
- Évaluation basique et seuils de sortie.

## V1.1

- OpenSearch pour BM25 avancé et filtres par champs.
- Alertes hebdomadaires par email.

## V2

- Intégration opendata (indicateurs avec graphiques).
- Filtres thématiques et taxonomie.
- Authentification basique.

## V3

- Graphe de connaissances ciblé.
  - Ontologie restreinte pour explicabilité.
- """,
- "documentation/07-roadmap/changelog.md": """"# Changelog

## [Unreleased]

- Initialisation de la documentation.

## v0.1.0 - MVP

- Ingestion PDF, indexation Qdrant + BM25 local.
- Pipeline RAG hybride + reranking.
- API FastAPI et UI simple.
- Évaluation et checklist V1.

## ===== 08-branding =====

"documentation/08-branding/naming.md": """"# Naming

- Nom produit : **Kaitiaki**
- Sens : gardien, protecteur, veilleur.
- Variantes dépôt :

- **kaitiaki**
- **kaitiaki-nc**

""",

"documentation/08-branding/ui-copy.md": """"# UI Copy

- Titre : Kaitiaki — Veille NC
- Champ de recherche : "Saisissez votre question"
- Bouton : "Chercher"
- Avertissement : "Kaitiaki fournit des synthèses d'informations publiques. Ce n'est pas un avis juridique."

- Messages d'erreur : "Aucune source trouvée pour cette requête", "Temps de réponse dépassé"

```
""",
}
```

def main() -> None:

parser = argparse.ArgumentParser(description="Génère l'arborescence documentation/ pour Kaitiaki.")

parser.add\_argument("--force", action="store\_true", help="Écrase les fichiers existants.")

args = parser.parse\_args()

python

```
created, skipped, overwritten = 0, 0, 0
```

```
for rel_path, content in FILES.items():
 out_path = ROOT / rel_path
 out_path.parent.mkdir(parents=True, exist_ok=True)
 if out_path.exists() and not args.force:
 skipped += 1
 continue
 if out_path.exists() and args.force:
 overwritten += 1
 else:
 created += 1
 out_path.write_text(content, encoding="utf-8")
```

```
print(f"Documentation générée sous: {DOC}")
```

```
print(f"Créés: {created} | Ignorés: {skipped} | Écrasés: {overwritten}")
```

if name == "main":

main()

perl

Souhaitez-vous également un script similaire pour initialiser un squelette `kaitiaki/` (code) avec quelques fichiers de base déjà prêts à exécuter, afin d'avoir le dépôt complet immédiatement opérationnel ?

```
::contentReference[oaicite:0]{index=0}
```