AYI NEDJIMI
Consultants

RAG Architecture | Guide - Guide Pratique Cybersecurite

Catégorie : Intelligence Artificielle   Lecture : 20 min   Publié le : 07/12/2025   Auteur : Ayi NEDJIMI

RAG (Retrieval Augmented Generation) : architecture, implémentation, cas d RAG Architecture | Guide Complet 2025. Expert en cybersécurité et.

Définition et origines

Plutôt que de s'appuyer uniquement sur les connaissances encodées durant le pré-entraînement (paramètres du modèle), le RAG **récupère dynamiquement des documents pertinents** depuis une base de connaissances externe, puis les injecte dans le contexte du LLM pour générer une réponse informée et factuelle. RAG (Retrieval Augmented Generation) : architecture, implémentation, cas d RAG Architecture | Guide Complet 2025. Expert en cybersécurité et. Dans un contexte où l'intelligence artificielle transforme les pratiques de cybersécurité, la maîtrise de ia rag retrieval augmented generation devient un avantage stratégique pour les équipes techniques. Nous abordons notamment : qu'est-ce que le rag ?, pourquoi utiliser le rag ? et architecture d'un système rag. Les professionnels y trouveront des recommandations actionnables, des commandes prêtes à l'emploi et des stratégies de mise en œuvre adaptées aux environnements d'entreprise.

Formule Conceptuelle du RAG

Réponse = LLM(Question + Documents_Récupérés)

Au lieu de : Réponse = LLM(Question)

Le problème que résout le RAG

Les LLM classiques (GPT-4, Claude, Mistral) souffrent de trois limitations majeures que le RAG adresse directement :

1. Hallucinations et informations erronées
Sans accès à des sources vérifiables, un LLM peut générer des informations plausibles mais fausses avec une confiance totale. Exemple : inventer des citations d'études inexistantes, des dates incorrectes, ou des procédures erronées.

Solution RAG : Le modèle base sa réponse sur des documents réels récupérés, réduisant les hallucinations de 40-60% selon les benchmarks.

2. Connaissances figées (knowledge cutoff)
GPT-4 (cutoff avril 2023) ne connaît rien des événements post-formation. Impossible de répondre sur la réglementation 2024, les nouveaux produits, ou les données internes d'entreprise.

Solution RAG : La base de connaissances est mise à jour indépendamment du modèle. Ajoutez un document aujourd'hui, interrogez-le demain.

Votre organisation est-elle prête à faire face aux attaques basées sur l'IA ?

3. Absence de traçabilité
Difficile de vérifier d'où provient une réponse LLM. Problématique pour les domaines réglementés (santé, finance, juridique) où la source doit être citée.

Solution RAG : Chaque réponse peut inclure les documents sources (titre, page, score de similarité), permettant une vérification humaine.

RAG vs Fine-tuning

Le RAG et le fine-tuning sont deux approches complémentaires pour adapter un LLM à un domaine spécifique. Voici leur comparaison détaillée :

Critère	RAG	Fine-tuning
Mise à jour des connaissances	Immédiate (ajout de documents)	Nécessite re-entraînement (semaines)
Coût initial	Faible ($100-500 setup)	Élevé ($5K-50K pour entraînement)
Coût d'usage	Tokens context longs ($0.01-0.03/ requête)	Identique modèle base
Traçabilité	Sources citables	Aucune (connaissances dans les poids)
Hallucinations	Réduites (groundé par documents)	Persistantes
Domaine d'application	Connaissances factuelles, Q&A, docs	Style, format, tâches spécialisées

Approche Hybride (Best Practice)

En production, combinez les deux : **fine-tuning pour le style/format** (ton, structure de réponse, termes métier) + **RAG pour les connaissances factuelles** (documentation, procédures, données évolutives).

Exemple : Un chatbot juridique fine-tuné sur le vocabulaire juridique français + RAG sur la base de jurisprudence actualisée.

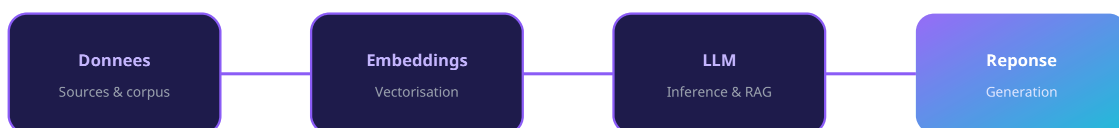
Schéma simplifié du fonctionnement

Voici le flux de données d'un système RAG en 5 étapes :

```
1. Question utilisateur : "Quelle est la procédure de remboursement ?"
   ↓
2. Conversion en embedding : [0.23, -0.45, 0.67, ...] (768 dimensions)
   ↓
3. Recherche vectorielle : Top 5 chunks similaires dans la base
   → Chunk #142 (score: 0.89) : "La procédure de remboursement..."
   → Chunk #87  (score: 0.82) : "Délais de traitement..."
   ↓
4. Injection dans le prompt LLM :
   "Contexte: [chunks récupérés]
    Question: Quelle est la procédure de remboursement ?
    Réponds en te basant uniquement sur le contexte fourni."
   ↓
5. Génération de la réponse + citation des sources
```

Temps de traitement typique : 200-800ms total (50ms retrieval + 150-750ms génération selon modèle et longueur).

Pipeline Intelligence Artificielle



Architecture IA - Du traitement des donnees a la generation de reponses

Notre avis d'expert
Chez Ayi NEDJIMI Consultants, nous constatons que la majorité des organisations sous-estiment les risques liés aux modèles de langage déployés en production. La sécurité des LLM ne se limite pas au prompt engineering : elle exige une approche systémique couvrant les embeddings, les pipelines de données et les mécanismes de contrôle d'accès aux API.

Pourquoi utiliser le RAG ?

Limiter les hallucinations

Le RAG réduit significativement les hallucinations en **groundant** (ancrant) les réponses du LLM dans des documents vérifiables. Les benchmarks montrent une réduction de 40-70% des hallucinations par rapport à un LLM vanilla.

Mécanismes anti-hallucination

- **Context grounding** : Le prompt système force le modèle à ne répondre que basé sur le contexte fourni

- **Citation explicite** : Demander au LLM de citer le passage exact utilisé
- **Verification step** : Un second appel LLM vérifie la cohérence entre réponse et sources
- **Fallback explicite** : Si aucun document pertinent (score < seuil), répondre "Information non disponible"

```
# Prompt anti-hallucination
system_prompt = """
Tu es un assistant qui répond UNIQUEMENT basé sur le contexte fourni.
Si la réponse n'est pas dans le contexte, réponds EXACTEMENT :
"Je n'ai pas trouvé cette information dans la documentation."

NE PAS inventer d'information. NE PAS utiliser tes connaissances générales.
"""
```

Métriques mesurables : Faithfulness score (fidélité aux sources) typiquement >0.85 avec RAG bien configuré vs 0.50-0.70 sans RAG.

Données à jour et spécifiques

Le principal avantage du RAG est la **séparation entre le modèle et les données**. Vous pouvez mettre à jour vos connaissances sans toucher au LLM.

Mise à jour en temps réel
Processus typique pour ajouter de nouvelles données :

1. **Upload** : Nouveau document déposé dans S3/dossier watched
2. **Parsing** : Extraction du texte (OCR si PDF scanné)
3. **Chunking** : Découpage en segments de 512 tokens avec overlap 50 tokens
4. **Embedding** : Conversion en vecteurs via API (OpenAI, Cohere, Mistral)
5. **Indexation** : Insertion dans la base vectorielle

Délai total : 30 secondes à 5 minutes selon volume (100 pages → ~2 min avec parallélisation).

Données propriétaires et métier
Le RAG excelle pour exploiter vos données internes que les LLM publics ne connaissent pas :

- Documentation technique interne (wikis, Confluence, Notion)
- Base de tickets support client (résolutions passées)
- Catalogues produits avec spécifications détaillées
- Procédures et réglementations d'entreprise
- Historiques de projets et post-mortems

Attention : Qualité des données

Garbage in, garbage out. Un RAG sur des documents obsolètes, contradictoires ou mal structurés produira des réponses médiocres. Prévoir un audit qualité des sources. Pour approfondir, consultez Phishing IA : Quand les Defenses Traditionnelles Echouent.

Traçabilité et sources

La traçabilité est critique dans les domaines réglementés (santé, finance, juridique) et pour la confiance utilisateur. Le RAG permet de **citer précisément les sources** utilisées pour générer chaque réponse.

Métadonnées exploitables
Chaque chunk stocké dans la base vectorielle peut inclure :

- **Document source** : titre, URL, path S3
- **Localisation** : page, section, paragraphe
- **Metadata** : date de publication, auteur, version, tags
- **Score de similarité** : 0.0-1.0 indiquant la pertinence

```
// Exemple de réponse avec sources
{
  "answer": "La période de remboursement standard est de 14 jours ouvrés...",
  "sources": [
    {
      "document": "Politique_Remboursement_v2.3.pdf",
      "page": 5,
      "chunk_text": "Article 3.2 - Délais : Le client dispose...",
      "similarity_score": 0.89,
      "url": "https://docs.company.com/policies/refund"
    },
    {
      "document": "FAQ_Client_2024.md",
      "section": "Remboursements",
      "similarity_score": 0.82
    }
  ]
}
```

Interface utilisateur
En production, afficher les sources avec :

- **Citations inline** : [1], [2] dans le texte avec références en bas
- **Accordéons** : Clic sur source → affichage du chunk complet
- **Liens directs** : Vers le document source (PDF, page X)
- **Indicateurs de confiance** : Badge "Haute confiance" si score >0.85

Coûts vs fine-tuning

Analyse économique détaillée pour un chatbot traitant **100K requêtes/mois** :

Poste de coût	RAG (GPT-4o)	Fine-tuning (GPT-4o-mini)
Setup initial	$500 (dev + infra)	$8K (préparation dataset + entraînement)
Embeddings (one-time)	$50 (10M tokens @ $0.13/1M)	N/A
Base vectorielle	$150/mois (Qdrant Cloud 1GB)	N/A
Retrieval per query	Négligeable (<1ms CPU)	N/A
LLM inference	$900/mois (3K tokens avg @ $2.50/1M input + $10/1M output)	$200/mois (modèle fine-tuné plus petit)
Total mois 1	$1,600	$8,200
Total mois 6	$6,300	$9,200
Mise à jour connaissances	$10 (embeddings incrémentaux)	$3K-8K (re-training)

ROI du RAG

Cas concret
En février 2024, une entreprise de Hong Kong a perdu 25 millions de dollars après qu'un employé a été trompé par un deepfake vidéo lors d'une visioconférence. Les attaquants avaient recréé l'apparence et la voix du directeur financier à l'aide de modèles d'IA générative, démontrant les risques concrets de cette technologie en contexte corporate.

Comment garantir que vos modèles de machine learning ne deviennent pas des vecteurs d'attaque ?

Le RAG est rentable dès le premier mois si vos données évoluent fréquemment (>1 mise à jour/trimestre). Le break-even vs fine-tuning se situe entre 4-6 mois selon le volume de requêtes.

Optimisations de coûts

- **Caching** : Redis pour queries fréquentes → -30-50% de coûts LLM
- **Modèle hybride** : GPT-4o-mini pour retrieval simple, GPT-4o pour questions complexes
- **Embeddings locaux** : Modèles open-source (all-MiniLM-L6-v2) → $0 après setup
- **Compression de contexte** : LLMLingua réduit tokens de 50% avec perte minimale de qualité

Cas d'usage idéaux

Le RAG excelle dans ces scénarios :

1. Support client intelligent

- **Problème** : Agents submergés par questions répétitives, temps de recherche dans la doc
- **Solution RAG** : Chatbot qui interroge base de connaissances (FAQ, tickets résolus, procédures)
- **Résultat** : -40% de tickets niveau 1, -60% temps de résolution, satisfaction +25%
- **Exemple** : Intercom, Zendesk AI utilisent du RAG en backend

2. Analyse de documentation technique

- **Problème** : Développeurs perdent 2-4h/jour à chercher dans la doc (APIs, specs, wikis)
- **Solution RAG** : Assistant qui index Confluence, GitHub wikis, Swagger/OpenAPI specs
- **Résultat** : Réponses en <30s vs 15-30min de recherche manuelle
- **Exemple** : GitHub Copilot Chat intègre RAG sur la documentation des repos

3. Q&A sur corpus juridique/réglementaire

- **Problème** : Textes de loi, jurisprudence, réglementations → milliers de pages
- **Solution RAG** : Recherche sémantique + extraction de clauses pertinentes
- **Résultat** : Juristes gagnent 10-20h/semaine, risques de non-conformité réduits
- **Exemple** : Outils comme Harvey AI, Robin AI pour cabinets d'avocats

4. Onboarding employés

- **Problème** : Nouveaux arrivants posent les mêmes questions (congés, notes de frais, outils)
- **Solution RAG** : Chatbot RH sur handbook employé, policies, guides internes
- **Résultat** : -50% de sollicitation RH, autonomie nouvelle recrue accélérée

5. Recommandations e-commerce

- **Problème** : "Je cherche un cadeau pour ma mère qui aime le jardinage" → requête complexe
- **Solution RAG** : Embeddings sur descriptions produits + historiques achats similaires
- **Résultat** : Taux de conversion +15-30% vs recherche mot-clé

Cas où le RAG n'est PAS adapté

- Tâches créatives pures (génération de poèmes, design)
- Raisonnement mathématique complexe (mieux : fine-tuning sur dataset math)
- Très faible volume de données (<50 documents → un LLM vanilla suffit)
- Besoin de réponses instantanées (<100ms) → coût du retrieval trop élevé

Architecture d'un système RAG

Vue d'ensemble de l'architecture

Un système RAG complet se décompose en **deux pipelines distincts** : indexation (offline) et requête (online).

```
┌─────────────────────────────────────────────────────────┐
│                    PIPELINE INDEXATION                  │
│                        (Offline)                        │
└─────────────────────────────────────────────────────────┘

 Documents sources (PDF, DOCX, HTML, Markdown)
    ↓
 Document Loader & Parser (PyPDF2, Unstructured)
    ↓
 Text Splitter / Chunker (512 tokens, overlap 50)
    ↓
 Embedding Model (OpenAI text-embedding-3-small)
    ↓
 Vector Database (Qdrant, Pinecone, Weaviate)

┌─────────────────────────────────────────────────────────┐
│                     PIPELINE REQUÊTE                    │
│                        (Online)                         │
└─────────────────────────────────────────────────────────┘

 Question utilisateur
    ↓
 Query Transformation (optional: expansion, réécriture)
    ↓
 Embedding de la query (même modèle que indexation)
    ↓
 Vector Search (top-k chunks, k=5-10)
    ↓
 Reranking (optional: cross-encoder)
    ↓
 Context Assembly (concaténation des chunks)
    ↓
 Prompt Engineering (system + context + question)
    ↓
 LLM Generation (GPT-4o, Claude 3.5, Mistral Large)
    ↓
 Response + Sources
```

Temps typiques : Indexation = 1-2 min pour 100 pages | Requête = 200-800ms end-to-end

Phase d'indexation (offline)

Cette phase s'exécute une fois au setup, puis de manière incrémentale à chaque ajout de documents. Elle transforme vos documents bruts en vecteurs interrogeables.

1. Chargement et parsing
Extraction du texte selon le format source :

- **PDF** : PyPDF2, pdfplumber (texte natif) | Tesseract (OCR si scan)
- **DOCX/PPTX** : python-docx, python-pptx
- **HTML** : BeautifulSoup, Trafilatura (extraction contenu principal)
- **Markdown** : Parsing direct avec métadonnées frontmatter
- **Code source** : Tree-sitter (parsing AST pour contexte)

2. Chunking strategy
Découpage du texte en segments cohérents. Paramètres critiques :

- **Taille de chunk** : 256-512 tokens (compromis granularité/contexte)
- **Overlap** : 10-20% pour éviter la perte d'information aux frontières
- **Stratégie** : Fixed-size | Semantic (selon paragraphes/sections) | Recursive

```
# Exemple LangChain
from langchain.text_splitter import RecursiveCharacterTextSplitter

text_splitter = RecursiveCharacterTextSplitter(
    chunk_size=512,
    chunk_overlap=50,
    separators=["\n\n", "\n", ".", " ", ""],
    length_function=len
)
chunks = text_splitter.split_documents(documents)
```

3. Génération des embeddings
Conversion de chaque chunk en vecteur haute dimension. Choix du modèle selon budget/performance :

Mise en pratique

Modèle	Dimensions	Coût	Performance
OpenAI text-embedding-3-small	1536	$0.02/1M tokens	⭐⭐⭐⭐
OpenAI text-embedding-3-large	3072	$0.13/1M tokens	⭐⭐⭐⭐⭐
Cohere embed-multilingual-v3	1024	$0.10/1M tokens	⭐⭐⭐⭐ (excellent multilangue)
Mistral embed	1024	$0.10/1M tokens	⭐⭐⭐⭐
all-MiniLM-L6-v2 (local)	384	Gratuit (self-hosted)	⭐⭐⭐

4. Stockage dans la base vectorielle
Insertion des embeddings avec métadonnées :

```
from qdrant_client import QdrantClient
from qdrant_client.models import PointStruct, VectorParams, Distance

client = QdrantClient("localhost", port=6333)

# Création de la collection
client.create_collection(
    collection_name="documentation",
    vectors_config=VectorParams(size=1536, distance=Distance.COSINE)
)

# Insertion des vecteurs
points = [
    PointStruct(
        id=i,
        vector=embedding,
        payload={
            "text": chunk.text,
            "source": chunk.metadata["source"],
            "page": chunk.metadata["page"],
            "created_at": "2024-01-15"
        }
    )
    for i, (chunk, embedding) in enumerate(zip(chunks, embeddings))
]

client.upsert(collection_name="documentation", points=points)
```

Performance indexation : 100K chunks en 15-30 min (avec parallélisation API embeddings)

Phase de requête (online)

Pipeline temps-réel déclenché à chaque question utilisateur. Objectif : **< 1 seconde end-to-end**.

1. Query transformation (optionnel)
Amélioration de la query avant recherche :

- **HyDE (Hypothetical Document Embeddings)** : Générer une réponse hypothétique, l'embedder, chercher avec
- **Query expansion** : Ajouter des termes synonymes/connexes
- **Multi-query** : Reformuler en 3-5 variantes, fusionner les résultats

2. Embedding de la query
Utiliser le **même modèle** que pour l'indexation (sinon incompatibilité des espaces vectoriels).

```
query_embedding = openai.embeddings.create(
    model="text-embedding-3-small",
    input="Quelle est la procédure de remboursement ?"
).data[0].embedding
```

3. Vector search
Recherche des k chunks les plus similaires (similarité cosine typiquement) :

```
results = client.search(
    collection_name="documentation",
    query_vector=query_embedding,
    limit=10,  # top-k
    score_threshold=0.7  # filtrer les résultats peu pertinents
)
```

Timing : 10-50ms avec index HNSW sur 1M vecteurs

4. Reranking (optionnel mais recommandé)
Les embeddings bi-encoders (dense vectors) sont rapides mais moins précis que les cross-encoders pour le ranking. Le reranking affine le top-10 en top-3 vraiment pertinent.

```
from sentence_transformers import CrossEncoder

reranker = CrossEncoder('cross-encoder/ms-marco-MiniLM-L-6-v2')

# Scorer chaque paire (query, chunk)
pairs = [[query, result.payload["text"]] for result in results]
scores = reranker.predict(pairs)

# Retrier selon les nouveaux scores
reranked = sorted(zip(results, scores), key=lambda x: x[1], reverse=True)[:3]
```

Gain : +10-20% de précision, +30-50ms de latence

5. Assembly du contexte
Concaténation des chunks dans le prompt avec séparateurs clairs :

```
context = "\n\n---\n\n".join([
    f"[Document: {r.payload['source']}, Page: {r.payload['page']}]\n{r.payload['text']}"
    for r in top_results
])
```

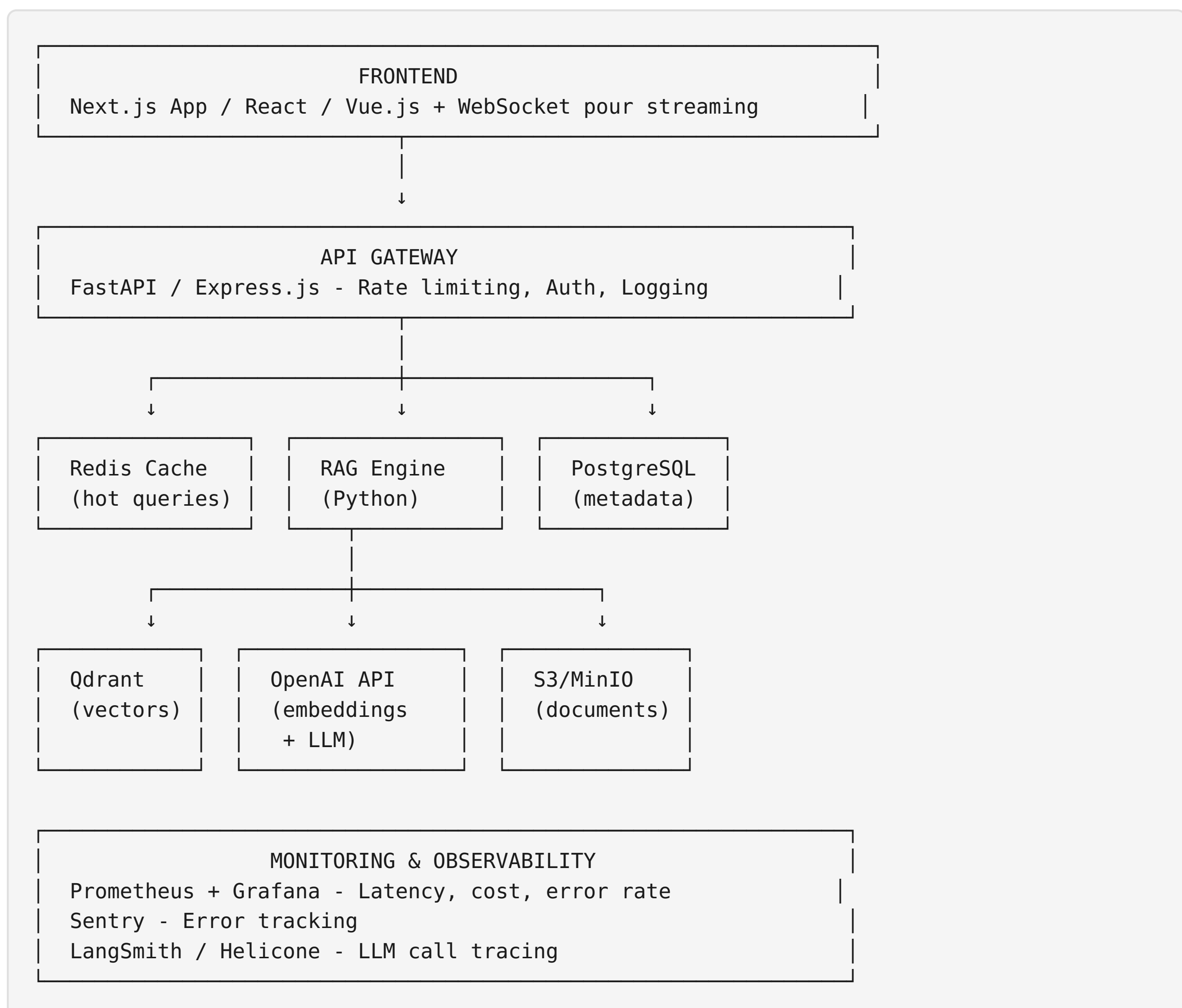
6. Génération LLM
Construction du prompt final et appel au LLM : Pour approfondir, consultez Reinforcement Learning Appliqué à la Cybersécurité.

```
prompt = f"""Tu es un assistant qui répond précisément basé sur le contexte fourni.

Contexte:
{context}

Question: {user_query}

Instructions:
- Réponds UNIQUEMENT avec les informations du contexte
- Cite les sources [Document: X, Page: Y]
- Si la réponse n'est pas dans le contexte, dis "Information non disponible"
"""

response = openai.chat.completions.create(
    model="gpt-4o",
    messages=[{"role": "user", "content": prompt}],
    temperature=0.1  # Réponse factuelle, peu créative
)
```

Temps génération : 150-750ms selon longueur de réponse et modèle

Diagramme d'architecture détaillé

Architecture production-ready avec haute disponibilité :



Redondance : Vector DB répliquée (3 nodes min), API stateless (scalable horizontalement), cache distribué.

Les composants clés

Document loader et parsing

Le loader extrait le contenu textuel des documents sources. LangChain fournit des loaders pré-construits pour 100+ formats.

```
from langchain.document_loaders import (
    PyPDFLoader,
    UnstructuredWordDocumentLoader,
    NotionDirectoryLoader,
    GitLoader,
    WebBaseLoader
)

# PDF
loader = PyPDFLoader("policy.pdf")
documents = loader.load()  # [Document(page_content="...", metadata={"source": ...,
"page": 1}), ...]

# Confluence / Notion (via export)
loader = NotionDirectoryLoader("notion_export/")

# GitHub repo
loader = GitLoader(
    clone_url="https://github.com/company/docs",
    branch="main",
    file_filter=lambda file_path: file_path.endswith(".md")
)
```

Pièges du parsing

- **PDF scannés** : Nécessite OCR (Tesseract, AWS Textract) → +5-30s/page
- **Tableaux** : Mal extraits par parsers basiques → utiliser Unstructured.io ou Camelot
- **Images avec texte** : Perdues si pas d'OCR → intégrer un modèle vision (GPT-4 Vision)

Chunking strategy

Le chunking est **l'optimisation la plus impactante** sur la qualité RAG. Mauvais chunking = mauvais retrieval = mauvaise réponse.

Stratégies courantes

Stratégie	Avantages	Inconvénients	Cas d'usage
Fixed-size	Simple, rapide	Coupe au milieu des phrases	Logs, tweets, texte court
Recursive	Respecte paragraphes/ sections	Chunks de taille variable	Documentation, articles
Semantic	Cohérence maximale	Coûteux (embeddings pour détecter breaks)	Textes narratifs, juridique
Structure-aware	Préserve hiérarchie (headers)	Nécessite parsing spécifique	Code source, XML/ JSON

Best Practice : Overlap + Metadata

Overlap de 10-20% pour éviter la perte d'information aux frontières. Ajoutez le titre de section comme métadonnée pour chaque chunk.

Exemple : Chunk = "... procédure complète..." + metadata["section"] = "3.2 Remboursements"

Paramètres optimaux (données empiriques)

- **Documentation technique** : 512 tokens, overlap 50
- **Support FAQ** : 256 tokens, overlap 25
- **Articles longs** : 1024 tokens, overlap 100
- **Code source** : Par fonction/classe (variable), overlap 20 lignes

Modèle d'embeddings

Le choix du modèle d'embeddings impacte performance et coûts. Comparaison sur benchmark MTEB (Massive Text Embedding Benchmark) :

Modèle	Score MTEB	Dimensions	Coût / Latence	Usage recommandé
OpenAI text-embedding-3-large	64.6	3072 (ou 256-3072)	$0.13/1M │ 100ms API	Production haute performance
OpenAI text-embedding-3-small	62.3	1536	$0.02/1M │ 80ms API	Meilleur rapport qualité/ prix
Cohere embed-multilingual-v3	66.8	1024	$0.10/1M │ 120ms API	Multilangue (100+ langues)
Voyage AI voyage-large-2	68.3	1536	$0.12/1M │ 90ms API	Top performance RAG
bge-large-en-v1.5 (local)	63.9	1024	Gratuit │ 20ms GPU local	Données sensibles, budget zéro

Règle d'or : Utiliser le MÊME modèle pour indexation ET requêtes. Sinon, incompatibilité des espaces vectoriels.

Base vectorielle

Comparaison des solutions leaders pour RAG :

Solution	Type	Forces	Faiblesses	Pricing
Qdrant	Open-source + Cloud	Rapide, filtering puissant, self-host	Moins d'intégrations que Pinecone	Gratuit (self) / $0.15/GB/mois cloud
Pinecone	Cloud managed	Setup en 5min, scalabilité auto	Vendor lock-in, coût élevé à scale	$0.096/GB/mois (s1 pods)
Weaviate	Open-source + Cloud	Modules pré-intégrés (OpenAI, Cohere)	Plus complexe à opérer	Gratuit (self) / $0.095/1M queries cloud
Chroma	Open-source	Ultra simple (pip install), local-first	Pas de cloud managed, scaling limité	Gratuit
pgvector (PostgreSQL)	Extension	Réutilise infra existante, ACID	Performance médiocre >1M vecteurs	Gratuit (si PostgreSQL déjà présent)

Recommandation Architecture

- **Prototype/MVP** : Chroma (local) ou Qdrant Cloud free tier
- **Production <1M vecteurs** : Qdrant self-hosted (ECS/GKE) ou Pinecone
- **Production >10M vecteurs** : Qdrant clusterisé ou Weaviate
- **Multimodal (texte+image)** : Weaviate avec modules vision

Retriever

Le retriever est le composant qui exécute la recherche vectorielle. Plusieurs stratégies existent :

1. Dense retrieval (standard)
Recherche par similarité cosine sur les embeddings. Avantage : rapide (10-50ms). Inconvénient : rate les matchs exacts de mots-clés rares.

2. Sparse retrieval (BM25)
Recherche lexicale traditionnelle. Excellent pour les acronymes, noms propres, identifiants techniques.

3. Hybrid retrieval (recommandé)
Combine dense + sparse avec fusion des scores. Gain de 10-25% de recall vs dense seul.

```
from langchain.retrievers import EnsembleRetriever
from langchain.retrievers import BM25Retriever
from langchain.vectorstores import Qdrant

# Dense retriever
vector_retriever = qdrant.as_retriever(search_kwargs={"k": 10})

# Sparse retriever
bm25_retriever = BM25Retriever.from_documents(documents)
bm25_retriever.k = 10

# Hybrid avec pondération 70% dense / 30% sparse
ensemble_retriever = EnsembleRetriever(
    retrievers=[vector_retriever, bm25_retriever],
    weights=[0.7, 0.3]
)
```

4. Self-querying retriever
Le LLM extrait des filtres structurés depuis la question naturelle. Exemple : "Documents sur le remboursement publiés en 2024" → filter: {"topic": "refund", "year": 2024}

LLM et prompt engineering

Le choix du LLM impacte qualité, latence et coûts. Benchmarks sur RAG tasks (HotpotQA, NQ) :

Modèle	Faithfulness	Answer Relevancy	Latence (avg)	Coût / 1K requêtes
GPT-4o	0.89	0.91	800ms	$9 (3K tokens)
GPT-4o-mini	0.84	0.87	600ms	$0.45
Claude 3.5 Sonnet	0.91	0.92	900ms	$9
Claude 3 Haiku	0.82	0.84	400ms	$0.75
Mistral Large 2	0.86	0.88	700ms	$6

Stratégie Routing LLM

Utilisez un modèle rapide/cheap (GPT-4o-mini, Haiku) pour 80% des queries simples, et escaladez vers GPT-4o/Claude 3.5 Sonnet pour les questions complexes (détectées par classification préalable). Réduction de coûts : 60-70%.

Template de prompt RAG production-ready

```
SYSTEM_PROMPT = """
Tu es un assistant expert qui répond aux questions en te basant STRICTEMENT sur le
contexte fourni.

RÈGLES OBLIGATOIRES :
1. Réponds UNIQUEMENT avec les informations présentes dans le contexte
2. Si l'information n'est pas dans le contexte, réponds : "Je n'ai pas trouvé cette
information dans la documentation disponible."
3. Cite TOUJOURS tes sources au format [Doc: nom_fichier, Page: X]
4. Sois précis et concis (maximum 3 paragraphes)
5. Si plusieurs documents se contredisent, mentionne les deux versions
"""

USER_PROMPT_TEMPLATE = """
Contexte extrait de la documentation :

{context}

---

Question de l'utilisateur : {question}

Réponse (avec citations des sources) :
"""
```

Implémentation pas à pas

Setup et dépendances

Installation des bibliothèques nécessaires pour un RAG complet :

```
# Core RAG
pip install langchain langchain-community langchain-openai

# Base vectorielle (choisir une)
pip install qdrant-client  # Qdrant
pip install pinecone-client  # Pinecone
pip install chromadb  # Chroma

# Document loaders
pip install pypdf unstructured python-docx

# Monitoring (optionnel)
pip install langsmith helicone-opentelemetry
```

Configuration des clés API :

```
# .env
OPENAI_API_KEY=sk-...
QDRANT_URL=https://your-cluster.qdrant.io
QDRANT_API_KEY=your-key
```

Étape 1 : Chargement des documents

```
from langchain.document_loaders import DirectoryLoader, PyPDFLoader
import os

# Charger tous les PDF d'un dossier
loader = DirectoryLoader(
    "./documents/",
    glob="**/*.pdf",
    loader_cls=PyPDFLoader,
    show_progress=True
)

documents = loader.load()
print(f"Chargé {len(documents)} pages depuis {len(set([d.metadata['source'] for d in
documents]))} fichiers")

# Exemple de document
# Document(
#     page_content="Procédure de remboursement...",
#     metadata={"source": "./documents/policy.pdf", "page": 5}
# )
```

Étape 2 : Chunking

```
from langchain.text_splitter import RecursiveCharacterTextSplitter

text_splitter = RecursiveCharacterTextSplitter(
    chunk_size=512,
    chunk_overlap=50,
    length_function=len,
    separators=["\n\n", "\n", ". ", " ", ""]
)

chunks = text_splitter.split_documents(documents)
print(f"{len(chunks)} chunks créés depuis {len(documents)} documents")

# Exemple : 1000 pages → ~5000 chunks de 512 tokens
```

Étape 3 : Génération des embeddings

```
from langchain.embeddings import OpenAIEmbeddings

# Initialiser le modèle d'embeddings
embeddings_model = OpenAIEmbeddings(
    model="text-embedding-3-small",
    dimensions=1536
)

# Les embeddings seront générés automatiquement lors de l'insertion
# dans la base vectorielle (voir étape suivante)
```

Étape 4 : Stockage dans la base vectorielle

```
from langchain.vectorstores import Qdrant
from qdrant_client import QdrantClient
from qdrant_client.models import Distance, VectorParams

# Connexion à Qdrant
client = QdrantClient(
    url=os.getenv("QDRANT_URL"),
    api_key=os.getenv("QDRANT_API_KEY")
)

# Création de la collection
collection_name = "documentation"

try:
    client.get_collection(collection_name)
    print(f"Collection {collection_name} existe déjà")
except:
    client.create_collection(
        collection_name=collection_name,
        vectors_config=VectorParams(size=1536, distance=Distance.COSINE)
    )
    print(f"Collection {collection_name} créée")

# Indexation des chunks (avec génération automatique des embeddings)
vectorstore = Qdrant.from_documents(
    documents=chunks,
    embedding=embeddings_model,
    url=os.getenv("QDRANT_URL"),
    api_key=os.getenv("QDRANT_API_KEY"),
    collection_name=collection_name,
    force_recreate=False  # Ne pas écraser si existe
)

print(f"Indexation terminée : {len(chunks)} vecteurs")
```

Étape 5 : Pipeline de requête

```
from langchain.chat_models import ChatOpenAI
from langchain.chains import RetrievalQA

# Initialiser le LLM
llm = ChatOpenAI(
    model="gpt-4o",
    temperature=0.1,  # Réponses factuelles
    max_tokens=500
)

# Créer le retriever
retriever = vectorstore.as_retriever(
    search_type="similarity",
    search_kwargs={"k": 5}  # Top 5 chunks
)

# Tester le retrieval
query = "Quelle est la procédure de remboursement ?"
relevant_docs = retriever.get_relevant_documents(query)

print(f"Trouvé {len(relevant_docs)} documents pertinents :")
for i, doc in enumerate(relevant_docs):
    print(f"\n[{i+1}] Source: {doc.metadata['source']}, Page: {doc.metadata.get('page',
'N/A')}")
    print(f"Extrait: {doc.page_content[:200]}...")
```

Étape 6 : Génération de la réponse

```
from langchain.prompts import PromptTemplate
from langchain.chains import RetrievalQA

# Prompt template personnalisé
prompt_template = """Tu es un assistant qui répond précisément basé sur le contexte
fourni.

Contexte:
{context}

Question: {question}

Instructions:
- Réponds UNIQUEMENT avec les informations du contexte
- Cite les sources (nom de fichier et page)
- Si la réponse n'est pas dans le contexte, dis "Information non disponible dans la
documentation"

Réponse:"""

PROMPT = PromptTemplate(
    template=prompt_template,
    input_variables=["context", "question"]
)

# Créer la chaîne RAG
qa_chain = RetrievalQA.from_chain_type(
    llm=llm,
    chain_type="stuff",  # Concatène tous les chunks dans le prompt
    retriever=retriever,
    return_source_documents=True,
    chain_type_kwargs={"prompt": PROMPT}
)

# Exécuter une requête
query = "Quelle est la procédure de remboursement ?"
result = qa_chain({"query": query})

print(f"Question: {query}")
print(f"\nRéponse: {result['result']}")
print(f"\nSources utilisées:")
for doc in result['source_documents']:
    print(f"- {doc.metadata['source']} (page {doc.metadata.get('page', 'N/A')})")
```

Code complet avec LangChain

Script complet production-ready avec gestion d'erreurs et logging :

```python
"""RAG Production-Ready avec LangChain + Qdrant + OpenAI"""
import os
from typing import List, Dict
import logging
from langchain.document_loaders import DirectoryLoader, PyPDFLoader
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain.embeddings import OpenAIEmbeddings
from langchain.vectorstores import Qdrant
from langchain.chat_models import ChatOpenAI
from langchain.chains import RetrievalQA
from langchain.prompts import PromptTemplate
from qdrant_client import QdrantClient
from qdrant_client.models import Distance, VectorParams
from dotenv import load_dotenv

load_dotenv()
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

class RAGSystem:
    def __init__(self):
        self.embeddings = OpenAIEmbeddings(model="text-embedding-3-small")
        self.llm = ChatOpenAI(model="gpt-4o", temperature=0.1)
        self.collection_name = "documentation"
        self.vectorstore = None

    def index_documents(self, docs_path: str):
        """Phase d'indexation : charge, chunke et indexe les documents"""
        logger.info(f"Chargement des documents depuis {docs_path}")

        # 1. Load
        loader = DirectoryLoader(docs_path, glob="**/*.pdf", loader_cls=PyPDFLoader)
        documents = loader.load()
        logger.info(f"{len(documents)} pages chargées")

        # 2. Chunk
        text_splitter = RecursiveCharacterTextSplitter(
            chunk_size=512, chunk_overlap=50
        )
        chunks = text_splitter.split_documents(documents)
        logger.info(f"{len(chunks)} chunks créés")

        # 3. Index dans Qdrant
        self.vectorstore = Qdrant.from_documents(
            documents=chunks,
            embedding=self.embeddings,
            url=os.getenv("QDRANT_URL"),
            api_key=os.getenv("QDRANT_API_KEY"),
            collection_name=self.collection_name
        )
        logger.info("Indexation terminée")

    def query(self, question: str, top_k: int = 5) -> Dict:
        """Phase de requête : recherche + génération"""
        if not self.vectorstore:
            # Reconnexion à une collection existante
            self.vectorstore = Qdrant(
                client=QdrantClient(
                    url=os.getenv("QDRANT_URL"),
                    api_key=os.getenv("QDRANT_API_KEY")
                ),
                collection_name=self.collection_name,
```

```
                embeddings=self.embeddings
            )

        retriever = self.vectorstore.as_retriever(
            search_kwargs={"k": top_k}
        )

        prompt = PromptTemplate(
            template="""Réponds précisément basé sur le contexte.

Contexte: {context}
Question: {question}

Réponse (avec sources):""",
            input_variables=["context", "question"]
        )

        qa_chain = RetrievalQA.from_chain_type(
            llm=self.llm,
            retriever=retriever,
            return_source_documents=True,
            chain_type_kwargs={"prompt": prompt}
        )

        result = qa_chain({"query": question})

        return {
            "answer": result["result"],
            "sources": [
                {
                    "file": doc.metadata["source"],
                    "page": doc.metadata.get("page"),
                    "excerpt": doc.page_content[:200]
                }
                for doc in result["source_documents"]
            ]
        }

# Usage
if __name__ == "__main__":
    rag = RAGSystem()

    # Phase 1 : Indexation (une fois)
    # rag.index_documents("./documents/")

    # Phase 2 : Requêtes
    result = rag.query("Quelle est la procédure de remboursement ?")
    print(f"Réponse: {result['answer']}")
    print(f"\nSources: {len(result['sources'])} documents")
```

Performance attendue : Indexation 100 pages = 2-3 min | Requête = 400-800ms Pour approfondir, consultez Top 10 des Attaques.

Patterns et variantes du RAG

Naive RAG

Le RAG naïf est l'implémentation la plus simple : **embed** → **search** → **concat** → **generate**. C'est le point de départ idéal pour un MVP.

Architecture

- Chunking fixed-size (512 tokens)
- Embeddings dense (OpenAI, Cohere)
- Recherche top-k par similarité cosine
- Concaténation brute des chunks dans le prompt
- Génération LLM directe

Limitations

- **Retrieval imprécis** : Pas de reranking → chunks peu pertinents dans le top-k
- **Contexte bruité** : Informations contradictoires ou redondantes
- **Pas de gestion des échecs** : Si retrieval rate, le LLM hallucine

Performance typique : Faithfulness 0.70-0.80, Answer Relevancy 0.75-0.85

Advanced RAG (avec reranking)

Améliore le Naive RAG avec des étapes de pré/post-traitement. Gain de performance : +10-20%.

Améliorations clés

1. **Query transformation** : Réécriture/expansion de la question avant recherche
2. **Hybrid search** : Dense + Sparse (BM25) avec fusion
3. **Reranking** : Cross-encoder pour affiner le top-k en top-3
4. **Context compression** : Supprimer les infos redondantes/non pertinentes
5. **Metadata filtering** : Filtrer par date, auteur, catégorie

```
from langchain.retrievers import ContextualCompressionRetriever
from langchain.retrievers.document_compressors import LLMChainExtractor

# Base retriever
base_retriever = vectorstore.as_retriever(search_kwargs={"k": 10})

# Compressor : extrait uniquement les passages pertinents
compressor = LLMChainExtractor.from_llm(llm)

compression_retriever = ContextualCompressionRetriever(
    base_compressor=compressor,
    base_retriever=base_retriever
)

# Les chunks retournés sont filtrés/compressés automatiquement
```

Performance : Faithfulness 0.82-0.88, Latency +100-200ms vs Naive

Modular RAG

Architecture flexible avec modules interchangeables. Permet d'adapter le pipeline selon le cas d'usage.

Modules disponibles

Module	Fonction	Implémentations
Query Processor	Transformer la query	HyDE, Multi-query, Decomposition
Retriever	Chercher documents	Dense, Sparse, Hybrid, Multi-vector
Reranker	Affiner le ranking	Cross-encoder, Cohere rerank, LLM-based
Context Builder	Assembler le contexte	Concatenation, Compression, Summarization
Generator	Générer réponse	GPT-4o, Claude, Mistral, Local (Llama)

Avantage : Tester facilement plusieurs configurations (A/B testing) sans refonte complète.

Self-RAG et Corrective RAG

Ces approches ajoutent des **mécanismes d'auto-correction** pour améliorer la fiabilité.

Self-RAG (Self-Reflective RAG)
Le LLM évalue lui-même la qualité du retrieval et de sa réponse :

1. **Retrieval necessity** : "Ai-je besoin de chercher des documents ou puis-je répondre directement ?"
2. **Relevance check** : "Les documents récupérés sont-ils pertinents ?"
3. **Support check** : "Ma réponse est-elle supportée par les documents ?"
4. **Utility check** : "Ma réponse répond-elle vraiment à la question ?"

```
# Pseudo-code Self-RAG
def self_rag(query, vectorstore, llm):
    # 1. Le LLM décide si retrieval est nécessaire
    needs_retrieval = llm.predict(f"Cette question nécessite-t-elle de chercher des documents ? {query}")

    if needs_retrieval:
        docs = vectorstore.search(query)

        # 2. Vérifie si les docs sont pertinents
        relevance_scores = [llm.score_relevance(doc, query) for doc in docs]
        relevant_docs = [doc for doc, score in zip(docs, relevance_scores) if score > 0.7]

        if not relevant_docs:
            return "Information non disponible"

        # 3. Génère réponse
        answer = llm.generate(query, relevant_docs)

        # 4. Vérifie support par les docs
        is_supported = llm.check_support(answer, relevant_docs)

        if not is_supported:
            # Retry avec query réécrite ou retourner "incertain"
            pass

    else:
        answer = llm.generate_direct(query)

    return answer
```

Corrective RAG (CRAG)

Si le retrieval initial échoue, le système essaie des stratégies alternatives :

- **Query rewriting** : Reformuler la question
- **Web search fallback** : Chercher sur internet (Tavily, Serper API)
- **Multi-source fusion** : Combiner base vectorielle + web + base SQL

Gain : Réduction des "Information non disponible" de 40-60%

Multi-hop RAG

Pour les questions complexes nécessitant plusieurs étapes de raisonnement. Exemple : "Qui est le CEO de l'entreprise qui a acquis Instagram ?"

Pipeline multi-hop

1. **Décomposition** : LLM décompose en sous-questions
   - "Quelle entreprise a acquis Instagram ?" → Réponse : Meta
   - "Qui est le CEO de Meta ?" → Réponse : Mark Zuckerberg
2. **Retrieval itératif** : Chaque sous-question déclenche une recherche
3. **Fusion** : Assembler les réponses intermédiaires
4. **Réponse finale** : Synthèse basée sur tous les hops

```
from langchain.chains import LLMChain
from langchain.prompts import PromptTemplate

# 1. Décomposer la question
decompose_prompt = PromptTemplate(
    template="Décompose cette question en étapes : {question}",
    input_variables=["question"]
)
decomposer = LLMChain(llm=llm, prompt=decompose_prompt)

steps = decomposer.run("Qui est le CEO de l'entreprise qui a acquis Instagram ?")
# Output: ["1. Quelle entreprise a acquis Instagram ?", "2. Qui est le CEO de cette
entreprise ?"]

# 2. Résoudre chaque étape
answers = []
for step in steps:
    docs = vectorstore.search(step)
    answer = llm.generate(step, docs)
    answers.append(answer)

# 3. Synthèse
final_answer = llm.synthesize(original_question, answers)
```

Cas d'usage : Analyse financière, recherche scientifique, investigations juridiques

Comparaison des approches

Approche	Complexité	Performance	Latence	Coût	Cas d'usage
Naive RAG	⭐	⭐⭐⭐	300ms	$	MVP, prototypes
Advanced RAG	⭐⭐	⭐⭐⭐⭐	500ms	$$	Production standard
Modular RAG	⭐⭐⭐	⭐⭐⭐⭐	500-800ms	$$	Multi-domaines, A/B testing
Self-RAG	⭐⭐⭐⭐	⭐⭐⭐⭐⭐	1-2s	$$$	Domaines critiques (santé, finance)
Multi-hop RAG	⭐⭐⭐⭐⭐	⭐⭐⭐⭐⭐	2-5s	$$$$	Questions complexes, analyse

Recommandation Progressive

Commencez par Naive RAG, mesurez la performance (faithfulness, answer relevancy). Si insuffisant, ajoutez progressivement : reranking (+10%) → hybrid search (+5%) → query transformation (+5%) → self-correction si besoin critique.

Optimisations avancées

Query transformation

La transformation de la query améliore le retrieval en enrichissant ou reformulant la question initiale.

1. HyDE (Hypothetical Document Embeddings)
Plutôt qu'embedder la question, générer une réponse hypothétique et l'embedder. Gain : +10-15% recall.

```
from langchain.chains import HypotheticalDocumentEmbedder

# Générer document hypothétique
hyde_prompt = f"""Génère une réponse plausible à cette question : {query}
(Pas besoin d'être factuel, juste vraisemblable)"""

hypothetical_doc = llm.predict(hyde_prompt)

# Embedder et chercher avec ce document
hyde_embedding = embeddings.embed_query(hypothetical_doc)
results = vectorstore.similarity_search_by_vector(hyde_embedding, k=5)
```

2. Multi-query
Générer 3-5 variantes de la question, chercher avec chacune, fusionner les résultats.

```
from langchain.retrievers.multi_query import MultiQueryRetriever

multi_query_retriever = MultiQueryRetriever.from_llm(
    retriever=vectorstore.as_retriever(),
    llm=llm
)

# Génère automatiquement des variantes et fusionne les résultats
docs = multi_query_retriever.get_relevant_documents(
    "Procédure de remboursement ?"
)
# Queries générées :
# - "Comment obtenir un remboursement ?"
# - "Quelle est la politique de retour ?"
# - "Délais de remboursement"
# - "Conditions de remboursement"
```

3. Step-back prompting
Pour une question spécifique, générer aussi une question plus générale pour capturer le contexte.

Exemple : "Quel est le taux d'imposition pour un revenu de 50K€ en 2024 ?" → Step-back : "Comment fonctionne le système d'imposition en France ?"

Hybrid search (dense + sparse)

Combine recherche vectorielle (sémantique) et lexicale (mots-clés exacts). **Gain : +15-25% recall** vs dense seul.

Pourquoi l'hybride est supérieur

Scénario	Dense (semantic)	Sparse (BM25)	Hybrid
"Procédure remboursement"	✅ Excellent	✅ Bon	✅ Excellent
"Article L1234-5" (code)	❌ Rate (pas de contexte)	✅ Parfait	✅ Parfait
"CEO de Meta"	⚠️ Peut confondre avec PDG, dirigeant	✅ Match exact sur CEO	✅ Meilleur des deux
"Remboursement" (ambigu)	✅ Comprend contexte	⚠️ Trop de résultats	✅ Filtré par sémantique

Implémentation avec Qdrant

```
from langchain.retrievers import EnsembleRetriever
from langchain_community.retrievers import BM25Retriever

# 1. Dense retriever (vectoriel)
vector_retriever = vectorstore.as_retriever(
    search_type="similarity",
    search_kwargs={"k": 10}
)

# 2. Sparse retriever (BM25)
# Extraire tous les documents pour construire l'index BM25
all_docs = [chunk.page_content for chunk in chunks]
bm25_retriever = BM25Retriever.from_texts(all_docs)
bm25_retriever.k = 10

# 3. Ensemble avec pondération
ensemble_retriever = EnsembleRetriever(
    retrievers=[vector_retriever, bm25_retriever],
    weights=[0.7, 0.3]  # 70% dense, 30% sparse
)

# Usage
docs = ensemble_retriever.get_relevant_documents("Article L1234-5")
```

Pondération optimale : 70/30 dense/sparse pour documentation générale, 50/50 pour textes juridiques/techniques.

Reranking avec cross-encoders

Les bi-encoders (embeddings) sont rapides mais moins précis. Les cross-encoders scorent chaque paire (query, doc) mais sont lents. Stratégie : **bi-encoder pour top-100 → cross-encoder pour affiner en top-3**.

Gain mesurable

- **Précision@3** : +12-20% vs sans reranking
- **Latence** : +30-80ms selon modèle
- **Coût** : Négligeable si modèle local (ms-marco-MiniLM)

```
from sentence_transformers import CrossEncoder
import numpy as np

# 1. Retrieval initial (top-10)
initial_results = vectorstore.similarity_search(query, k=10)

# 2. Reranking avec cross-encoder
reranker = CrossEncoder('cross-encoder/ms-marco-MiniLM-L-6-v2')

pairs = [[query, doc.page_content] for doc in initial_results]
scores = reranker.predict(pairs)

# 3. Retrier
ranked_indices = np.argsort(scores)[::-1]  # Descending
reranked_docs = [initial_results[i] for i in ranked_indices[:3]]

print(f"Top 3 après reranking :")
for i, (doc, score) in enumerate(zip(reranked_docs, sorted(scores, reverse=True)[:3])):
    print(f"{i+1}. Score: {score:.3f} | {doc.page_content[:100]}...")
```

Modèles recommandés

- **ms-marco-MiniLM-L-6-v2** : Rapide (20ms/query), bon compromis
- **bge-reranker-large** : Meilleure précision, multilangue
- **Cohere Rerank API** : $2/1K requêtes, excellent mais coûteux

Contextual compression

Réduit la taille du contexte en extrayant uniquement les passages pertinents. **Bénéfices** : -40-60% tokens, -30% latence LLM, -40% coûts.

```
from langchain.retrievers.document_compressors import (
    LLMChainExtractor,
    EmbeddingsFilter
)
from langchain.retrievers import ContextualCompressionRetriever

# Option 1 : Filtre par embeddings (rapide)
embeddings_filter = EmbeddingsFilter(
    embeddings=embeddings,
    similarity_threshold=0.76  # Garder uniquement si >76% similaire
)

# Option 2 : Extraction LLM (précis mais lent)
llm_extractor = LLMChainExtractor.from_llm(llm)

compression_retriever = ContextualCompressionRetriever(
    base_compressor=embeddings_filter,  # ou llm_extractor
    base_retriever=vectorstore.as_retriever(search_kwargs={"k": 10})
)

compressed_docs = compression_retriever.get_relevant_documents(query)
# Résultat : 3-5 docs au lieu de 10, avec uniquement les phrases pertinentes
```

Metadata filtering

Filtrer les documents avant ou après recherche vectorielle selon métadonnées. **Use case** : "Documents publiés après 2023" ou "Dans la catégorie Finance".

```
# Indexation avec métadonnées
for chunk in chunks:
    chunk.metadata["category"] = extract_category(chunk)  # "HR", "Finance", "IT"
    chunk.metadata["date"] = extract_date(chunk)  # "2024-01-15"
    chunk.metadata["confidentiality"] = "public"  # "public", "internal", "confidential"

vectorstore.add_documents(chunks)

# Recherche avec filtres
from qdrant_client.models import Filter, FieldCondition, MatchValue, Range

filter_condition = Filter(
    must=[
        FieldCondition(
            key="metadata.category",
            match=MatchValue(value="Finance")
        ),
        FieldCondition(
            key="metadata.date",
            range=Range(gte="2024-01-01")  # Depuis 2024
        )
    ]
)

results = vectorstore.similarity_search(
    query,
    k=5,
    filter=filter_condition
)
```

Best practice : Combinez filtering + recherche vectorielle plutôt que filtrer après (plus efficace). Pour approfondir, consultez Évaluation de LLM : Métriques, Benchmarks et Frameworks.

Caching intelligent

Le caching réduit drastiquement coûts et latence pour les queries répétitives. **Gain typique** : -50-70% coûts LLM, -80% latence.

Stratégies de caching

Niveau	Quoi cacher	Durée	Hit rate typique
Embeddings	Cache query embedding	1h-24h	20-40%
Retrieval	Cache top-k docs pour query	15min-1h	30-50%
LLM response	Cache réponse complète	5min-30min	40-60%
Semantic cache	Questions similaires → même réponse	1h-24h	50-70%

Implémentation avec Redis

```
import redis
import hashlib
import json

redis_client = redis.Redis(host='localhost', port=6379, decode_responses=True)

def cached_rag_query(query: str, ttl: int = 1800):  # 30 min
    # 1. Générer clé de cache
    cache_key = f"rag:{hashlib.md5(query.encode()).hexdigest()}"

    # 2. Vérifier cache
    cached = redis_client.get(cache_key)
    if cached:
        print("Cache HIT")
        return json.loads(cached)

    # 3. Exécuter RAG si cache MISS
    print("Cache MISS - Exécution RAG")
    result = rag_chain({"query": query})

    # 4. Sauvegarder dans cache
    redis_client.setex(
        cache_key,
        ttl,
        json.dumps(result)
    )

    return result

# Usage
result = cached_rag_query("Procédure de remboursement ?")
```

Semantic caching avancé

Plutôt que matcher exactement, utiliser la similarité vectorielle pour trouver des questions similaires.

```
from langchain.cache import RedisSemanticCache

# Cache sémantique : questions similaires partagent la réponse
semantic_cache = RedisSemanticCache(
    redis_url="redis://localhost:6379",
    embedding=embeddings,
    score_threshold=0.85  # Si similarité >85%, réutiliser réponse
)

# Questions traitées comme identiques :
# "Comment obtenir un remboursement ?"
# "Procédure pour se faire rembourser ?"
# "Je veux un remboursement, comment faire ?"
```

Passage en production

Métriques à surveiller

Le monitoring d'un système RAG nécessite des métriques spécifiques. Voici les KPIs essentiels :

Métriques de performance RAG

Métrique	Description	Seuil recommandé	Mesure
Faithfulness	Réponse fidèle aux sources	>0.85	LLM-as-judge ou annotation humaine
Answer Relevancy	Réponse pertinente à la question	>0.80	Similarité question-réponse
Context Precision	Chunks récupérés pertinents	>0.70	% de chunks utiles pour la réponse
Context Recall	Information nécessaire récupérée	>0.75	% d'info ground-truth dans contexte
Latency P95	95% des requêtes	<2s	Temps end-to-end

Métriques business

- **Resolution rate** : % de questions résolues sans escalade humaine (objectif: >80%)
- **User satisfaction** : Score moyen thumbs up/down (objectif: >4/5)
- **Deflection rate** : % de tickets support évités grâce au chatbot
- **Cost per query** : Coût total / nombre de requêtes

```
# Exemple monitoring avec RAGAS
from ragas import evaluate
from ragas.metrics import faithfulness, answer_relevancy, context_precision

# Dataset d'évaluation
eval_dataset = {
    "question": ["Procédure de remboursement ?"],
    "answer": ["La procédure est décrite dans..."],
    "contexts": [["Le remboursement s'effectue...", "Délais de 14 jours..."]],
    "ground_truths": ["Pour un remboursement, veuillez..."]
}

# Évaluation
result = evaluate(
    dataset=eval_dataset,
    metrics=[faithfulness, answer_relevancy, context_precision]
)

print(f"Faithfulness: {result['faithfulness']:.3f}")
print(f"Answer Relevancy: {result['answer_relevancy']:.3f}")
```

Gestion des coûts API

Les coûts API peuvent exploser rapidement en production. Stratégies d'optimisation :

1. Optimisation des modèles

- **Router intelligent** : GPT-4o-mini pour 70% des queries simples, GPT-4o pour les complexes
- **Embeddings locaux** : bge-large-en-v1.5 auto-hébergé vs $0.02/1M OpenAI
- **Context compression** : Réduire tokens de 40-60% avec LLMLingua

2. Budgets et limites

```
import openai
from datetime import datetime
import redis

class CostController:
    def __init__(self, daily_budget: float = 100.0):
        self.daily_budget = daily_budget
        self.redis = redis.Redis()

    def check_budget(self, estimated_cost: float) -> bool:
        today = datetime.now().strftime("%Y-%m-%d")
        spent_key = f"daily_spend:{today}"
        current_spend = float(self.redis.get(spent_key) or 0)

        if current_spend + estimated_cost > self.daily_budget:
            return False  # Budget dépassé

        return True

    def record_spend(self, cost: float):
        today = datetime.now().strftime("%Y-%m-%d")
        spent_key = f"daily_spend:{today}"
        self.redis.incrbyfloat(spent_key, cost)
        self.redis.expire(spent_key, 86400)  # 24h

def safe_llm_call(prompt: str, cost_controller: CostController):
    # Estimer coût (approximatif)
    estimated_tokens = len(prompt.split()) * 1.3  # Rule of thumb
    estimated_cost = (estimated_tokens / 1000) * 0.03  # $0.03/1K tokens

    if not cost_controller.check_budget(estimated_cost):
        return "Budget journalier atteint. Veuillez ressayer demain."

    response = openai.chat.completions.create(
        model="gpt-4o-mini",
        messages=[{"role": "user", "content": prompt}]
    )

    cost_controller.record_spend(estimated_cost)
    return response.choices[0].message.content
```

3. Analyse des coûts

Tracking détaillé par endpoint :

- **Embeddings** : $0.02/1M tokens (one-time pour indexation)
- **LLM génération** : $3-15/1K requêtes selon modèle et longueur
- **Reranking API** : $2/1K si Cohere (gratuit si local)

- **Vector DB** : $150-500/mois selon taille

Latence et optimisation

Décomposition de la latence typique (pour 1 requête) :

Étape	Latence	Optimisation
Query embedding	50-120ms	Cache (Redis) ou modèle local
Vector search	10-50ms	Index HNSW optimisé, SSD rapide
Reranking	30-100ms	Modèle local, GPU inference
LLM generation	200-800ms	Streaming, modèles plus petits
Total	290-1070ms	Parallélisation, caching agressif

Optimisations de latence

1. **Streaming** : Commencer à afficher la réponse dès les premiers tokens
2. **Parallélisation** : Exécuter embedding + retrieval en parallèle si possible
3. **Predictive prefetching** : Pré-calculer pour questions fréquentes
4. **Edge deployment** : Déployer la vector DB près des utilisateurs

Gestion des mises à jour de documents

Stratégies pour maintenir la base de connaissances à jour :

1. Indexation incrémentale

```
import hashlib
from datetime import datetime

class IncrementalIndexer:
    def __init__(self, vectorstore):
        self.vectorstore = vectorstore
        self.doc_hashes = {}  # Tracking des versions

    def compute_hash(self, document_path: str) -> str:
        with open(document_path, 'rb') as f:
            return hashlib.md5(f.read()).hexdigest()

    def needs_update(self, document_path: str) -> bool:
        current_hash = self.compute_hash(document_path)
        old_hash = self.doc_hashes.get(document_path)
        return current_hash != old_hash

    def update_document(self, document_path: str):
        if not self.needs_update(document_path):
            print(f"{document_path} - Pas de changement")
            return

        print(f"Mise à jour de {document_path}")

        # 1. Supprimer anciens chunks
        self.vectorstore.delete(filter={"source": document_path})

        # 2. Recharger et ré-indexer
        new_chunks = self.load_and_chunk(document_path)
        self.vectorstore.add_documents(new_chunks)

        # 3. Mettre à jour hash
        self.doc_hashes[document_path] = self.compute_hash(document_path)

        print(f"Indexé {len(new_chunks)} nouveaux chunks")
```

2. Pipeline CI/CD pour docs

Automatiser les mises à jour via webhooks :

- **Git hooks** : Push sur branch docs → déclenche re-indexation
- **Confluence/Notion webhooks** : Page modifiée → export + update automatique
- **S3 events** : Nouveau fichier uploadé → indexation
- **Scheduled updates** : Scan quotidien des sources externes

3. Versioning et rollback

Maintenir plusieurs versions de l'index pour permettre rollback en cas de problème :

```
# Collections versionnées
collections = {
    "docs_v1": "2024-01-15",
    "docs_v2": "2024-01-20",  # Current
    "docs_v3": "2024-01-22"   # Staging
}

# Swap instantané en cas de problème
def rollback_to_previous():
    current = "docs_v2"
    previous = "docs_v1"
    # Update config to point to previous version
    update_config("active_collection", previous)
```

Monitoring et alerting

Stack de monitoring production-ready :

Métriques techniques

- **Prometheus + Grafana** : Métriques de latence, throughput, erreurs
- **LangSmith / Helicone** : Tracing des appels LLM, coûts en temps réel
- **Vector DB monitoring** : CPU, mémoire, taille index, QPS

Alertes critiques

```
# Exemple alertes Prometheus
groups:
  - name: rag_alerts
    rules:
      - alert: HighLatency
        expr: histogram_quantile(0.95, rag_query_duration_seconds) > 2
        for: 5m
        annotations:
          summary: "RAG latency P95 > 2s"

      - alert: LowFaithfulness
        expr: avg_over_time(rag_faithfulness_score[10m]) < 0.8
        for: 5m
        annotations:
          summary: "Faithfulness score dropping"

      - alert: HighCosts
        expr: increase(llm_api_cost_total[1h]) > 50
        annotations:
          summary: "LLM costs >$50/hour"
```

Scalabilité

Architecture pour supporter la montée en charge :

Scaling horizontal

- **API stateless** : Load balancer devant multiples instances FastAPI
- **Vector DB clusterisée** : Qdrant 3+ nodes avec réplication
- **Cache distribué** : Redis Cluster pour haute disponibilité
- **Queue async** : Celery + RabbitMQ pour indexation background

Limites par composant

Composant	Single node	Cluster	Bottleneck
Qdrant	10K QPS	100K+ QPS	Disk I/O pour >50M vecteurs
OpenAI API	500 RPM	10K+ RPM (tier 5)	Rate limits, tokens/min
FastAPI app	1K RPS	Illimité	CPU pour reranking local

Checklist pré-production

Liste de vérification avant mise en production :

✅ **Performance & Qualité**

- ☐ Faithfulness >0.85 sur dataset de test (100+ questions)
- ☐ Answer relevancy >0.80
- ☐ Latence P95 <2 secondes
- ☐ Load testing 10x le trafic attendu
- ☐ A/B test vs baseline existant

✅ **Sécurité & Compliance**

- ☐ Authentification utilisateurs (OAuth, JWT)
- ☐ Rate limiting par user (100 requêtes/min)
- ☐ Logs audités (qui, quand, quoi)
- ☐ Données sensibles masquées/chiffrées
- ☐ RGPD compliance si UE

✅ **Ops & Monitoring**

- ☐ Métriques Prometheus configurées
- ☐ Alertes Slack/email opérationnelles
- ☐ Backup quotidien vector DB
- ☐ Procédure rollback testée
- ☐ Documentation déploiement à jour

✅ **Business**

- ☐ Budget API validé (avec marge 50%)
- ☐ SLA défini (99.5% uptime)
- ☐ Plan de support utilisateurs
- ☐ Métriques business trackées
- ☐ Escalation path si problème

Sources et références : ArXiv IA · Hugging Face Papers

Questions fréquentes

Quel LLM utiliser pour un RAG ?

Le choix dépend de votre budget et exigences qualité :

- **Production premium** : GPT-4o ou Claude 3.5 Sonnet (faithfulness >0.90, $9/1K requêtes)
- **Production standard** : GPT-4o-mini (excellent compromis, $0.45/1K requêtes)
- **Français natif** : Mistral Large 2 (optimisé pour le français, $6/1K requêtes)
- **Budget limité** : Claude 3 Haiku (rapide et bon marché, $0.75/1K requêtes)
- **On-premise** : Llama 3.1 70B ou Mistral 8x22B (auto-hébergé)

Recommandation : Commencez avec GPT-4o-mini, évaluez sur vos données, puis montez en gamme si nécessaire.

Combien de chunks récupérer (top-k) ?

Le top-k optimal dépend de votre stratégie :

- **Sans reranking** : k=3-5 (trop de bruit si plus élevé)
- **Avec reranking** : récupérer k=10-20, reranker en top-3
- **Questions simples** : k=3 suffit souvent
- **Questions complexes** : k=5-8 pour plus de contexte

Test empirique : Évaluez avec différents k sur votre dataset. Généralement k=5 est un bon départ.

Limite technique : Le context window du LLM. GPT-4o (128K tokens) peut gérer 200+ chunks de 512 tokens, mais la performance baisse au-delà de 10-15 chunks pertinents.

Le RAG fonctionne-t-il pour toutes les langues ?

Oui, mais avec des nuances importantes :

Langues excellemment supportées

- **Anglais** : Performance maximale (tous les modèles)
- **Français, Allemand, Espagnol** : Très bon avec OpenAI, Cohere, Mistral
- **Chinois, Japonais** : Bon avec OpenAI, excellent avec modèles spécialisés

Recommandations par langue

- **Français** : Mistral embed + Mistral Large 2 (natif) ou OpenAI (universel)
- **Multilingue** : Cohere embed-multilingual-v3 + GPT-4o
- **Langue rare** : Tester avec mBERT embeddings + GPT-4o

Piège : Mélanger les langues dans un même corpus réduit la performance. Séparez par langue ou utilisez des embeddings multilingues dédiés.

Comment gérer la confidentialité des données ?

Plusieurs approches selon votre niveau d'exigence :

Niveau 1 : Cloud public (Standard)

- OpenAI API + Qdrant Cloud
- Données transitées en HTTPS, chiffrées au repos
- Politique de rétention : 30 jours max chez OpenAI
- **Adapté pour** : Données publiques, support client général

Niveau 2 : Cloud dédié (Sensible)

- Azure OpenAI (dans votre tenant) + Qdrant self-hosted
- Contrôle total des données, logs audités
- Chiffrement avec clés maîtrisées
- **Adapté pour** : Données internes, finance, santé

Niveau 3 : On-premise (Confidentiel)

- Llama 3.1 70B + embeddings locaux + Qdrant local
- Zero data exfiltration, air-gapped possible
- Coût : $50K+ setup + expertise DevOps
- **Adapté pour** : Défense, bancaire, propriété intellectuelle

Bonnes pratiques

- **Anonymisation** : Remplacer noms/emails par tokens avant indexation
- **Access control** : Filtrer les documents selon permissions utilisateur
- **Audit trail** : Logger qui accède à quoi, quand
- **Data residency** : Choisir région cloud selon contraintes légales

Peut-on faire du RAG sans base vectorielle ?

Oui, plusieurs alternatives existent, avec des trade-offs :

1. Recherche lexicale pure (BM25/Elasticsearch)

- **Avantages** : Setup simple, matches exacts parfaits, moins cher
- **Inconvénients** : Pas de compréhension sémantique, synonymes ratés
- **Usage** : Documentation technique avec beaucoup d'acronymes

2. PostgreSQL avec pg_trgm

- **Avantages** : Réutilise infrastructure existante, recherche floue
- **Inconvénients** : Performance dégrade >1M documents
- **Usage** : POC, petites bases de connaissances

3. RAG "stateless" (contextuel)

- Passer directement tous les documents pertinents dans le context
- **Limite** : Context window LLM (128K tokens = ~200 pages max)
- **Usage** : Analyse ponctuelle de quelques documents

4. Hybrid avec SQL

```
-- Recherche SQL traditionnelle
SELECT title, content, ts_rank(search_vector, query) as score
FROM documents
WHERE search_vector @@ to_tsquery('remboursement & procedure')
ORDER BY score DESC
LIMIT 5;
```

Verdict : Les bases vectorielles offrent la meilleure expérience RAG pour >90% des cas. Les alternatives sont valables pour des besoins très spécifiques ou des contraintes techniques fortes.

Ressources open source associées :

- CUDAEmbeddings — Serveur d'embeddings GPU (Python)
- CyberSec-Assistant-3B — LLM cybersécurité généraliste (HuggingFace)
- rag-langchain-fr — Dataset RAG & LangChain (HuggingFace)

Ayi NEDJIMI Consultants — Expert cybersécurité offensive & intelligence artificielle

ayinedjimi-consultants.fr · ayi@ayinedjimi-consultants.fr

© 2025 — Reproduction interdite sans autorisation.