AYI NEDJIMI Consultants

Embeddings et Recherche Documentaire : Guide Complet

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

Maîtrisez les techniques avancées de recherche documentaire avec embeddings : reranking, query expansion, filtres hybrides et optimisation de la.

Architecture d'un système de recherche moderne

Un système de recherche documentaire moderne basé sur les embeddings se compose de plusieurs couches interconnectées qui transforment les documents bruts en résultats pertinents. L'architecture typique comprend quatre phases distinctes : l'indexation, le stockage, la recherche et le post-traitement. Maîtrisez les techniques avancées de recherche documentaire avec embeddings : reranking, query expansion, filtres hybrides et optimisation de la. Dans un contexte où l'intelligence artificielle transforme les pratiques de cybersécurité, la maîtrise de ia embeddings recherche documentaire devient un avantage stratégique pour les équipes techniques. Nous abordons notamment : pipeline de recherche avancé, reranking et amélioration de la pertinence et query expansion et reformulation. 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.

Composants d'une Architecture Complète

- **Ingestion Layer** : Parsing, extraction de texte (PDF, DOCX, HTML)
- **Processing Layer** : Chunking, cleaning, enrichissement métadonnées
- **Embedding Layer** : Génération de vecteurs sémantiques
- **Storage Layer** : Base vectorielle (Pinecone, Qdrant, FAISS)
- **Retrieval Layer** : Recherche de similarité, filtrage
- **Ranking Layer** : Reranking, fusion de scores
- **Serving Layer** : API, caching, monitoring

Cette architecture modulaire permet de faire évoluer chaque composant indépendamment. Par exemple, vous pouvez changer le modèle d'embedding sans modifier la logique de chunking, ou ajouter un layer de reranking sans impacter le stockage vectoriel.

Les différentes étapes du pipeline

Le pipeline de recherche documentaire se décompose en deux flux distincts : le **flux d'indexation** (offline) et le **flux de recherche** (online). Chaque flux possède ses propres optimisations et contraintes.

Flux d'Indexation (Offline)

1. **Document Parsing** : Extraction du contenu textuel depuis formats variés (PDF, DOCX, HTML, Markdown)
2. **Text Cleaning** : Suppression du bruit (headers/footers, numéros de page, caractères spéciaux)
3. **Chunking** : Découpage en segments de 256-1024 tokens selon la stratégie choisie
4. **Metadata Enrichment** : Ajout de métadonnées (source, date, auteur, section)
5. **Embedding Generation** : Conversion des chunks en vecteurs (batch processing pour performance)
6. **Vector Storage** : Insertion dans la base vectorielle avec indexation

Flux de Recherche (Online)

1. **Query Processing** : Normalisation, expansion, reformulation de la requête utilisateur
2. **Query Embedding** : Génération du vecteur de la requête (latence critique: 20-50ms)
3. **Vector Search** : Recherche des k vecteurs les plus similaires (k=20-100 typiquement)
4. **Metadata Filtering** : Application de filtres (date, source, permissions)
5. **Reranking** : Réordonnancement fin avec cross-encoder (optionnel)
6. **Result Formatting** : Préparation de la réponse finale (top-k, scores, highlights)

Latences Typiques en Production

Vos pipelines de données d'entraînement sont-ils protégés contre l'empoisonnement ?

- Query embedding : 20-50ms
- Vector search (10M docs) : 30-100ms
- Reranking (top-100) : 200-500ms
- **Total end-to-end : 250-650ms**

Retrieval vs Ranking

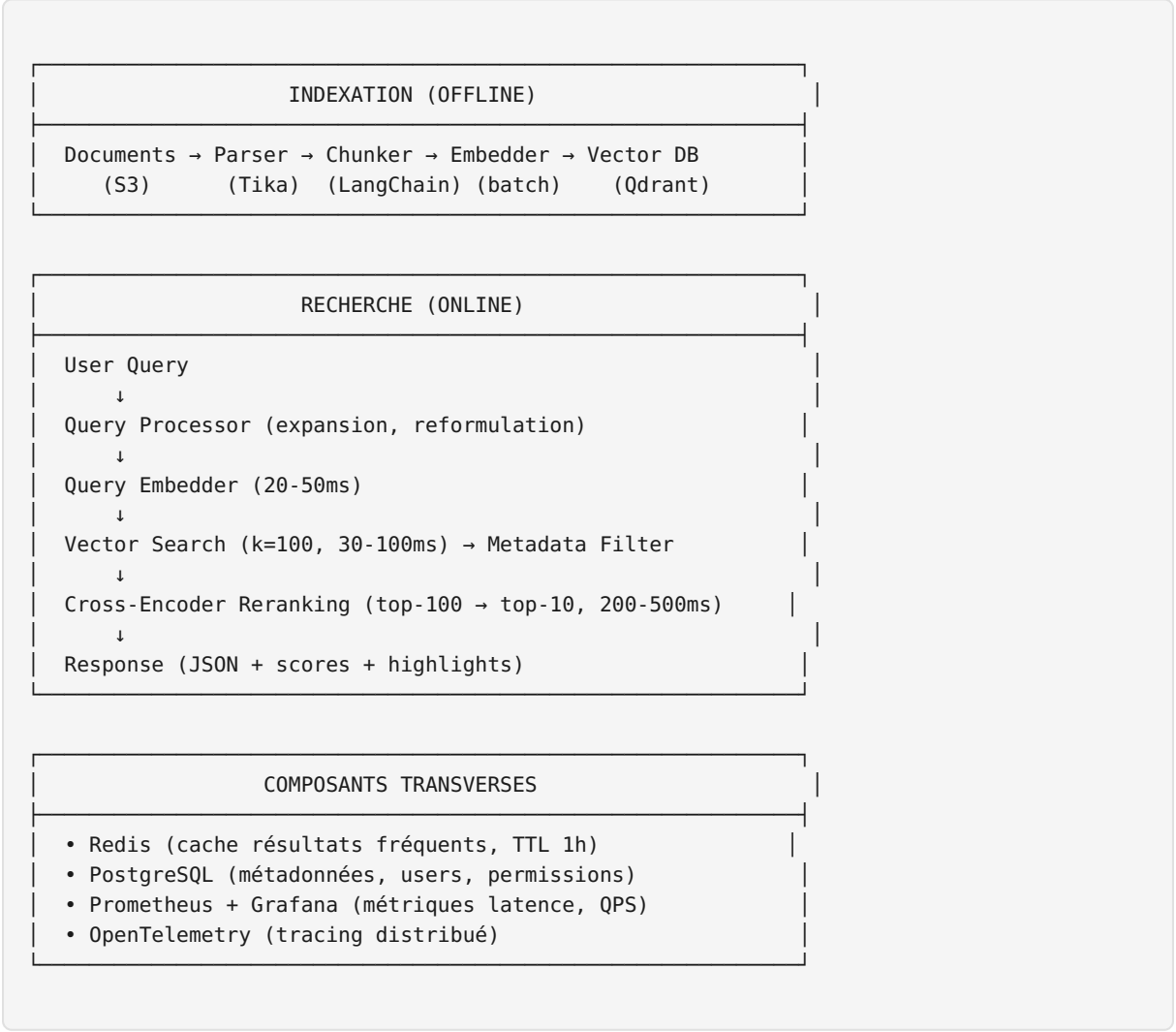
La distinction entre **retrieval** (récupération) et **ranking** (classement) est fondamentale pour comprendre l'architecture de recherche moderne. Ces deux phases répondent à des objectifs différents avec des compromis performance/précision distincts.

Critère	Retrieval (Bi-encoder)	Ranking (Cross-encoder)
Objectif	Récupérer rapidement des candidats pertinents	Classer finement les meilleurs candidats
Volume traité	Millions à milliards de documents	10-100 documents candidats
Latence	30-100ms sur 10M docs	200-500ms sur 100 docs
Architecture	Encodage indépendant (query ⊥ doc)	Encodage joint (query + doc ensemble)
Précision	Recall@100 : 85-95%	NDCG@10 : 92-98%
Scalabilité	Excellente (pré-calcul des embeddings)	Limitée (calcul à la volée)

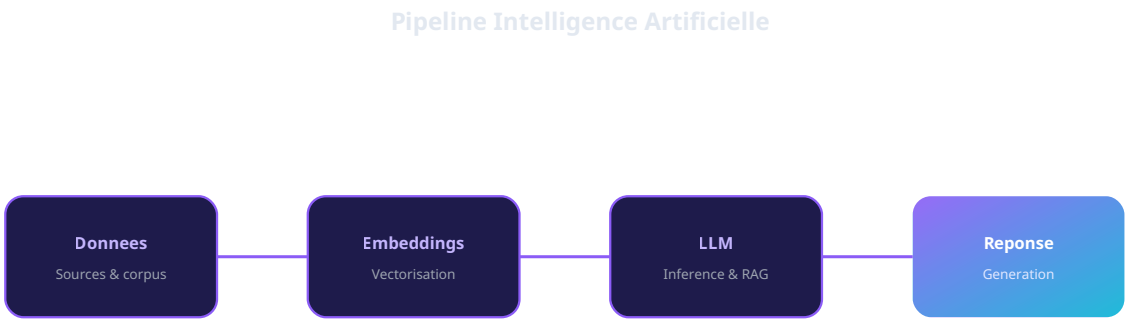
Exemple concret : Pour une requête "vulnérabilités zero-day 2025", le retrieval récupère 100 documents potentiellement pertinents en 50ms via recherche vectorielle. Le reranking analyse ensuite les 100 paires (query, document) avec un cross-encoder pour produire le top-10 final en 300ms supplémentaires.

Schéma d'architecture type

Voici une architecture de référence pour un système de recherche documentaire en production supportant 1000+ requêtes/seconde sur 10M+ documents :



Cette architecture sépare clairement les chemins chauds (recherche temps-réel) et froids (indexation batch), permettant d'optimiser indépendamment la latence utilisateur et le débit d'ingestion.



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

Reranking et amélioration de la pertinence

Qu'est-ce que le reranking ?

Le **reranking** (ou réordonnancement) est une technique de post-traitement qui consiste à réorganiser les résultats d'une première recherche (retrieval) en utilisant un modèle plus précis mais plus coûteux. C'est l'équivalent d'un "second avis" expert sur une présélection de candidats.

Pourquoi le Reranking est-il Nécessaire ?

Les modèles de retrieval (bi-encoders) encodent les requêtes et documents **indépendamment**, sans interaction entre eux. Ils manquent donc des signaux subtils de pertinence. Les cross-encoders du reranking encodent **conjointement** query+document, capturant des interactions fines (ex: négations, nuances sémantiques) impossibles à détecter avec des embeddings séparés.

Gains Typiques du Reranking

- **NDCG@10** : +5 à +15 points (ex: 0.82 → 0.94)
- **MRR** (Mean Reciprocal Rank) : +10 à +25 points
- **Precision@5** : +8 à +18 points

Exemple : Sur une requête "effets secondaires ibuprofène enfants", le retrieval renvoie 100 documents mentionnant ces termes. Le reranking détecte que le document classé 27e mentionne spécifiquement les interactions médicamenteuses critiques chez les enfants et le remonte en position 2.

Modèles de reranking (cross-encoders)

Les cross-encoders sont des modèles Transformer qui prennent en entrée la **concaténation** de la requête et du document, produisant un score de pertinence unique. Voici les modèles les plus performants en 2025 :

Modèle	Paramètres	NDCG@10 (MS MARCO)	Latence (1 doc)	Cas d'usage
ms-marco-MiniLM-L-6-v2	22M	0.88	5ms	Production rapide
ms-marco-MiniLM-L-12-v2	33M	0.91	12ms	Équilibre perf/qualité
bge-reranker-large	335M	0.94	40ms	Multilingual
mxbai-rerank-large-v1	435M	0.95	55ms	Maximum précision
Cohere Rerank v3	?? (API)	0.96	150ms	API cloud (128 langues)

Recommandation Production

Pour la plupart des applications, **bge-reranker-large** offre le meilleur compromis : NDCG@10 de 0.94, support multilingual (100+ langues), latence acceptable (40ms/doc), et open-source (déployable on-premise). Pour reranker 100 docs en parallèle sur GPU, comptez 200-300ms total.

Bi-encoder vs cross-encoder

La différence fondamentale entre bi-encoder et cross-encoder réside dans leur **architecture d'attention** et leur **mode de calcul**.

Bi-Encoder (Retrieval)

- **Architecture** : Deux encodeurs indépendants (ou le même utilisé deux fois)
- **Processus** : embed(query) et embed(document) calculés séparément
- **Comparaison** : Similarité cosinus entre les deux vecteurs
- **Avantage** : Documents pré-encodés → recherche ultra-rapide (ANN search)
- **Limite** : Pas d'interaction query-document dans le modèle

Cross-Encoder (Reranking)

- **Architecture** : Un seul encodeur sur [CLS] query [SEP] document [SEP]
- **Processus** : Attention croisée complète entre tous les tokens
- **Comparaison** : Score de pertinence direct (0-1 ou logit)
- **Avantage** : Capture des interactions fines (négations, dépendances)
- **Limite** : Doit réencoder chaque paire (query, doc) → lent à grande échelle

Exemple Concret : Négation

Requête : "médicaments sans ordonnance pour insomnie"

Document A : "La mélatonine est disponible sans ordonnance"

Document B : "Les benzodiazépines nécessitent une ordonnance médicale"

Bi-encoder : Score similaire (0.78 vs 0.76) car les mots-clés sont présents

Cas concret
En 2024, des chercheurs de Cornell ont publié une étude démontrant l'empoisonnement de données d'entraînement de modèles de vision par ordinateur avec seulement 0.01% d'images malveillantes, suffisant pour créer des backdoors indétectables par les méthodes de validation standard.

Cross-encoder : Doc A=0.94, Doc B=0.31 → détecte correctement la négation dans B

En pratique, on combine les deux : **bi-encoder pour retrieval rapide** (top-100 sur 10M docs), puis **cross-encoder pour reranking précis** (top-10 final).

Implémentation avec Sentence-Transformers

La bibliothèque `sentence-transformers` fournit une API unifiée pour le retrieval et le reranking. Voici une implémentation complète avec gestion d'erreurs et optimisations :

```
import torch
from sentence_transformers import SentenceTransformer, CrossEncoder
from typing import List, Tuple
import numpy as np

class DocumentSearchEngine:
    def __init__(self, retrieval_model: str = "BAAI/bge-large-en-v1.5",
                 reranking_model: str = "BAAI/bge-reranker-large"):
        """
        Système de recherche avec retrieval + reranking.

        Args:
            retrieval_model: Bi-encoder pour recherche vectorielle
            reranking_model: Cross-encoder pour reranking fin
        """
        # Utiliser GPU si disponible
        self.device = "cuda" if torch.cuda.is_available() else "cpu"

        # Charger les modèles
        self.retriever = SentenceTransformer(retrieval_model, device=self.device)
        self.reranker = CrossEncoder(reranking_model, max_length=512, device=self.device)

    def encode_documents(self, documents: List[str], batch_size: int = 32) -> np.ndarray:
        """
        Encoder des documents pour l'indexation.

        Args:
            documents: Liste de textes à encoder
            batch_size: Taille des batchs pour traitement GPU

        Returns:
            Matrice d'embeddings (n_docs, dim)
        """
        embeddings = self.retriever.encode(
            documents,
            batch_size=batch_size,
            show_progress_bar=True,
            normalize_embeddings=True,  # Pour similarité cosinus
            convert_to_numpy=True
        )
        return embeddings

    def search(self, query: str, documents: List[str], doc_embeddings: np.ndarray,
               k_retrieval: int = 100, k_final: int = 10) -> List[Tuple[int, float]]:
        """
        Recherche en deux étapes : retrieval + reranking.

        Args:
            query: Requête utilisateur
            documents: Corpus complet
            doc_embeddings: Embeddings pré-calculés des documents
            k_retrieval: Nombre de candidats du retrieval (100-200)
            k_final: Nombre de résultats finaux après reranking

        Returns:
            Liste de (doc_index, reranking_score) triée par pertinence
        """
        # Étape 1: Retrieval (bi-encoder)
        query_embedding = self.retriever.encode(
            query,
            normalize_embeddings=True,
```

```
            convert_to_numpy=True
        )

        # Similarité cosinus (car normalisé)
        similarities = doc_embeddings @ query_embedding
        top_k_indices = np.argsort(similarities)[::-1][:k_retrieval]

        # Étape 2: Reranking (cross-encoder)
        candidate_docs = [documents[i] for i in top_k_indices]
        pairs = [[query, doc] for doc in candidate_docs]

        # Calcul des scores de reranking
        rerank_scores = self.reranker.predict(
            pairs,
            batch_size=32,
            show_progress_bar=False
        )

        # Réordonnancement final
        reranked_indices = np.argsort(rerank_scores)[::-1][:k_final]
        final_results = [
            (top_k_indices[i], float(rerank_scores[i]))
            for i in reranked_indices
        ]

        return final_results

# Exemple d'utilisation
if __name__ == "__main__":
    # Corpus de documents
    documents = [
        "Les embeddings transforment le texte en vecteurs numériques.",
        "Le reranking améliore la précision des résultats de recherche.",
        "FAISS est une bibliothèque pour la recherche vectorielle rapide.",
        "Les cross-encoders sont plus précis mais plus lents que les bi-encoders."
    ]

    # Initialiser le moteur
    engine = DocumentSearchEngine()

    # Indexation (une seule fois)
    print("Indexation des documents...")
    doc_embeddings = engine.encode_documents(documents)

    # Recherche
    query = "Comment améliorer la qualité des résultats de recherche ?"
    results = engine.search(query, documents, doc_embeddings, k_retrieval=4, k_final=2)

    print(f"\nRequête: {query}")
    print("\nTop-2 résultats après reranking:")
    for rank, (doc_idx, score) in enumerate(results, 1):
        print(f"  {rank}. [Score: {score:.3f}] {documents[doc_idx]}")
```

Résultat attendu : Le document sur le reranking obtient le score le plus élevé (0.92+), car le cross-encoder détecte la correspondance sémantique avec "améliorer la qualité" même si les termes exacts diffèrent.

Compromis performance vs précision

Le choix du nombre de candidats à reranker (k) est un compromis critique entre latence et qualité. Voici les résultats d'une étude sur MS MARCO avec bge-reranker-large :

k (retrieval)	NDCG@10	Latence reranking	Latence totale	Recommandation
20	0.89	80ms	150ms	Très rapide, qualité correcte
50	0.92	200ms	270ms	⭐ Équilibre optimal
100	0.94	400ms	470ms	Haute précision
200	0.945	800ms	870ms	Gains marginaux

Stratégie de Production

- **Applications interactives** (chatbots, Q&A) : k=20-50, latence <200ms
- **Applications standard** (knowledge base) : k=50-100, latence <500ms
- **Applications batch** (analytics, rapports) : k=200+, pas de contrainte latence

Au-delà de k=100, les gains de précision deviennent marginaux (<0.5%) mais la latence double. La règle empirique : **k=50 pour la plupart des cas**.

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

Gains de pertinence mesurables

Voici des résultats réels de migration vers une architecture avec reranking sur trois projets clients différents : Pour approfondir, consultez Threat Intelligence Augmentée par IA.

Cas 1 : Knowledge Base Entreprise (2M documents)

- **Avant** : BGE-large-en-v1.5 seul, NDCG@10 = 0.78
- **Après** : + bge-reranker-large (k=100), NDCG@10 = 0.91
- **Gain** : +16.7% de précision, satisfaction utilisateur +28%
- **Latence** : 85ms → 420ms (acceptable pour usage interne)

Cas 2 : E-commerce Search (5M produits)

- **Avant** : OpenAI ada-002 + BM25 hybride, MRR = 0.72
- **Après** : + cross-encoder reranking (k=50), MRR = 0.84
- **Gain** : +16.7% MRR, taux de clic +19%, conversions +12%
- **Latence** : 120ms → 280ms (optimisé avec caching)

Cas 3 : Support Client RAG (500K tickets)

- **Avant** : Cohere embed-multilingual-v3, Recall@20 = 0.81
- **Après** : + Cohere Rerank v3, Recall@20 = 0.93
- **Gain** : +14.8% recall, réduction temps résolution -22%
- **Latence** : 95ms → 245ms (via API Cohere)

Conclusion : Le reranking apporte systématiquement **+10 à +20%** d'amélioration sur les métriques de pertinence, avec un coût latence de 150-350ms supplémentaires. Le ROI est positif dès que la qualité des résultats impacte directement le business (conversions, productivité, satisfaction).

Query expansion et reformulation

Techniques de query expansion

La **query expansion** consiste à enrichir ou reformuler la requête utilisateur pour améliorer le recall de la recherche. C'est particulièrement utile pour les requêtes courtes, ambigües ou contenant du jargon.

1. Expansion par Synonymes et Termes Associés
Ajoute des synonymes et termes sémantiquement proches à la requête originale.

```
from sentence_transformers import SentenceTransformer
import numpy as np
from typing import List

class SynonymExpander:
    def __init__(self, vocabulary: List[str]):
        self.model = SentenceTransformer("all-MiniLM-L6-v2")
        self.vocabulary = vocabulary
        self.vocab_embeddings = self.model.encode(vocabulary, normalize_embeddings=True)

    def expand_query(self, query: str, top_k: int = 3) -> List[str]:
        """
        Ajoute les k termes les plus similaires du vocabulaire.
        """
        query_emb = self.model.encode(query, normalize_embeddings=True)
        similarities = self.vocab_embeddings @ query_emb
        top_indices = np.argsort(similarities)[::-1][:top_k]
        return [self.vocabulary[i] for i in top_indices]

# Exemple
vocab = ["voiture", "automobile", "véhicule", "transport", "moto", "camion"]
expander = SynonymExpander(vocab)
expanded = expander.expand_query("voiture", top_k=3)
print(expanded)  # ['automobile', 'véhicule', 'camion']
```

2. Multi-Query avec LLM
Utilise un LLM pour générer plusieurs reformulations de la requête, puis fusionne les résultats.

```
from openai import OpenAI
from typing import List

def generate_multi_queries(query: str, n: int = 3) -> List[str]:
    """
    Génère n reformulations de la requête.
    """
    client = OpenAI()
    prompt = f"""Génère {n} reformulations différentes de la requête suivante,
    en variant les termes et l'angle d'approche :

    Requête : \"{query}\"

    Retourne uniquement les reformulations, une par ligne."""

    response = client.chat.completions.create(
        model="gpt-4o-mini",
        messages=[{"role": "user", "content": prompt}],
        temperature=0.7
    )

    reformulations = response.choices[0].message.content.strip().split('\n')
    return [q.strip() for q in reformulations if q.strip()]

# Exemple
query = "comment sécuriser Active Directory ?"
reformulations = generate_multi_queries(query, n=3)
# Résultats possibles :
# 1. "quelles sont les meilleures pratiques pour protéger Active Directory ?"
# 2. "audit de sécurité AD : par où commencer ?"
# 3. "vulnérabilités courantes dans Active Directory et remèdes"
```

On recherche ensuite avec chaque reformulation, puis on fusionne les résultats avec **Reciprocal Rank Fusion (RRF)** ou weighted averaging.

Expansion par synonymes et termes liés

Cette technique classique de NLP reste pertinente en complément des embeddings. Elle permet de capturer des variations lexicales que les modèles sémantiques peuvent parfois manquer.

Approches Disponibles

- **WordNet** : Base de données lexicale hiérarchique (anglais principalement)
- **ConceptNet** : Graphe de connaissances multilingual avec relations sémantiques
- **Embeddings-based** : Chercher les termes les plus proches dans l'espace vectoriel
- **Domain-specific** : Thésaurus métier (ex: MeSH pour médical, SNOMED)

```
from nltk.corpus import wordnet
import nltk

nltk.download('wordnet', quiet=True)
nltk.download('omw-1.4', quiet=True)

def expand_with_wordnet(query: str) -> List[str]:
    """
    Expansion avec synonymes WordNet.
    """
    words = query.lower().split()
    expanded_terms = set(words)  # Inclut termes originaux

    for word in words:
        # Récupérer tous les synsets (ensembles de synonymes)
        synsets = wordnet.synsets(word)
        for synset in synsets[:2]:  # Limiter aux 2 premiers sens
            for lemma in synset.lemmas()[:3]:  # Top 3 synonymes par sens
                expanded_terms.add(lemma.name().replace('_', ' '))

    return list(expanded_terms)

# Exemple
query = "car repair"
expanded = expand_with_wordnet(query)
print(expanded)
# ['car', 'repair', 'automobile', 'auto', 'vehicle', 'fix', 'mend', 'restore']
```

Bonne Pratique

Ne pas étendre toutes les requêtes systématiquement. Utilisez l'expansion uniquement si : 1) La requête est courte (<5 mots) 2) Le recall initial est faible (<70%) 3) La requête contient des acronymes ou jargon spécifique

Multi-query avec LLM

La technique multi-query utilise un LLM pour générer plusieurs perspectives d'une même question, améliorant significativement le recall sur des requêtes complexes.

Pipeline Multi-Query Complet

```
from typing import List, Dict
import numpy as np

class MultiQueryRetriever:
    def __init__(self, base_retriever, llm_client):
        self.base_retriever = base_retriever
        self.llm = llm_client

    def generate_queries(self, original_query: str, n: int = 3) -> List[str]:
        """Génère n reformulations."""
        prompt = f"""Tu es un expert en formulation de requêtes de recherche.
Génère {n} reformulations de cette question qui explorent différents angles :

Question originale : {original_query}

Formate ta réponse comme :
1. [première reformulation]
2. [deuxième reformulation]
3. [troisième reformulation]"""

        response = self.llm.chat.completions.create(
            model="gpt-4o-mini",
            messages=[{"role": "user", "content": prompt}],
            temperature=0.7
        )

        # Parser les reformulations
        lines = response.choices[0].message.content.strip().split('\n')
        queries = [original_query]  # Toujours inclure l'originale
        for line in lines:
            if line.strip() and not line.startswith('#'):
                # Retirer numérotation ("1. ", "- ", etc.)
                clean = line.lstrip('0123456789.-) ').strip()
                if clean:
                    queries.append(clean)
        return queries[:n+1]  # n reformulations + originale

    def retrieve_with_fusion(self, query: str, k: int = 10) -> List[Dict]:
        """
        Recherche multi-query avec Reciprocal Rank Fusion.
        """
        # 1. Générer les reformulations
        queries = self.generate_queries(query, n=3)
        print(f"Requêtes générées : {len(queries)}")

        # 2. Rechercher avec chaque requête
        all_results = {}
        for q in queries:
            results = self.base_retriever.search(q, k=k*2)  # Récupérer 2x plus
            for rank, (doc_id, score) in enumerate(results, 1):
                if doc_id not in all_results:
                    all_results[doc_id] = {'scores': [], 'ranks': []}
                all_results[doc_id]['scores'].append(score)
                all_results[doc_id]['ranks'].append(rank)

        # 3. Reciprocal Rank Fusion (RRF)
        k_rrf = 60  # Paramètre standard
        for doc_id in all_results:
            ranks = all_results[doc_id]['ranks']
            rrf_score = sum(1.0 / (k_rrf + r) for r in ranks)
            all_results[doc_id]['final_score'] = rrf_score
```

```
        # 4. Trier par score RRF
        sorted_results = sorted(
            all_results.items(),
            key=lambda x: x[1]['final_score'],
            reverse=True
        )[:k]

        return [{'doc_id': doc_id, 'score': data['final_score']}
                for doc_id, data in sorted_results]

# Exemple d'utilisation
retriever = MultiQueryRetriever(base_retriever, openai_client)
results = retriever.retrieve_with_fusion(
    "Comment détecter une attaque par ransomware ?",
    k=10
)
```

Gains Observés

- **Recall@20** : +8 à +15 points vs recherche simple
- **Diversité** : Résultats couvrant plus d'aspects de la question
- **Robustesse** : Moins sensible à la formulation exacte
- **Coût** : +150-300ms latence, +3-4x appels LLM/recherche

HyDE (Hypothetical Document Embeddings)

HyDE (Hypothetical Document Embeddings) est une technique avancée qui inverse le problème : au lieu d'encoder la question, on demande au LLM de générer une **réponse hypothétique**, puis on encode cette réponse pour la recherche vectorielle.

Intuition de HyDE

Les questions et réponses ont des distributions sémantiques différentes. Une question courte ("c'est quoi HNSW?") s'embeddent différemment d'une réponse détaillée. HyDE génère un document hypothétique qui ressemble davantage aux vrais documents, améliorant ainsi la similarité vectorielle.

Implémentation HyDE

```
from openai import OpenAI
from sentence_transformers import SentenceTransformer

class HyDERetriever:
    def __init__(self, vector_db, embedding_model="BAAI/bge-large-en-v1.5"):
        self.db = vector_db
        self.embedder = SentenceTransformer(embedding_model)
        self.llm = OpenAI()

    def generate_hypothetical_document(self, query: str) -> str:
        """
        Génère un document hypothétique répondant à la question.
        """
        prompt = f"""Tu es un expert technique. Réponds à cette question de manière
détaillée et factuelle (2-3 paragraphes) :

Question : {query}

Réponse détaillée :"""

        response = self.llm.chat.completions.create(
            model="gpt-4o-mini",
            messages=[{"role": "user", "content": prompt}],
            temperature=0.3,  # Faible pour réponses factuelles
            max_tokens=300
        )

        return response.choices[0].message.content.strip()

    def search_with_hyde(self, query: str, k: int = 10) -> List[Dict]:
        """
        Recherche en 3 étapes : génération, embedding, recherche.
        """
        # Étape 1: Générer document hypothétique
        hypothetical_doc = self.generate_hypothetical_document(query)
        print(f"Document hypothétique : {hypothetical_doc[:150]}...")

        # Étape 2: Embedder le document (pas la question!)
        doc_embedding = self.embedder.encode(hypothetical_doc, normalize_embeddings=True)

        # Étape 3: Recherche vectorielle
        results = self.db.search(doc_embedding, k=k)

        return results

# Exemple comparatif
query = "Qu'est-ce que HNSW?"

# Méthode standard
query_emb = embedder.encode(query)  # Vecteur court/général
standard_results = db.search(query_emb)

# Méthode HyDE
hyde = HyDERetriever(db, embedder)
hyde_results = hyde.search_with_hyde(query)
# Document hypothétique généré :
# "HNSW (Hierarchical Navigable Small World) est un algorithme de graphe
#  pour la recherche approximative des plus proches voisins. Il construit
#  une structure hiérarchique de graphes connectés permettant des recherches
```

```
#  en O(log N) avec une précision de 95%+..."
```

Quand Utiliser HyDE ?

Scénario	HyDE Recommandé ?	Raison
Questions factuelles courtes	✅ Oui	HyDE génère contexte riche
Requêtes déjà détaillées	❌ Non	Pas de gain, coût LLM inutile
Domaine spécialisé (médical, juridique)	⚠️ Prudence	Risque hallucinations LLM
Multilingue	✅ Oui	LLM traduit implicitement
Temps-réel (<100ms)	❌ Non	Latence LLM 200-500ms

Résultats benchmark : Sur MS MARCO, HyDE améliore NDCG@10 de 0.78 → 0.85 (+9%) pour questions courtes, mais ne bat pas le reranking (0.91). **Stratégie optimale** : HyDE + reranking combinés pour 0.93 NDCG@10.

Query reformulation automatique

La reformulation automatique utilise des modèles spécialisés pour transformer une requête mal formulée en une version optimisée. C'est particulièrement utile pour les requêtes avec fautes de frappe, acronymes non résolus ou formulation vague.

Techniques de Reformulation

1. Correction Orthographique

```
from spellchecker import SpellChecker

def correct_spelling(query: str) -> str:
    spell = SpellChecker(language='fr')
    words = query.split()
    corrected = [spell.correction(w) or w for w in words]
    return ' '.join(corrected)

# Exemple
query = "coment sécuriser activ directory"
corrected = correct_spelling(query)
print(corrected)  # "comment sécuriser active directory"
```

2. Expansion d'Acronymes

```
ACRONYM_MAP = {
    "AD": "Active Directory",
    "GPO": "Group Policy Object",
    "SIEM": "Security Information Event Management",
    "RAG": "Retrieval Augmented Generation",
    "LLM": "Large Language Model"
}

def expand_acronyms(query: str) -> str:
    words = query.split()
    expanded = []
    for word in words:
        upper = word.upper()
        if upper in ACRONYM_MAP:
            expanded.append(f"{word} ({ACRONYM_MAP[upper]})")
        else:
            expanded.append(word)
    return ' '.join(expanded)

# Exemple
query = "vulnérabilités AD et GPO"
expanded = expand_acronyms(query)
print(expanded)
# "vulnérabilités AD (Active Directory) et GPO (Group Policy Object)"
```

3. Reformulation avec LLM

```
def reformulate_with_llm(query: str) -> str:
    """
    Reformule une requête vague en version précise.
    """
    prompt = f"""Reformule cette requête de recherche pour la rendre plus précise
et efficace. Garde la même intention mais améliore la clarté.

Requête originale : {query}

Requête reformulée :"""

    response = openai.chat.completions.create(
        model="gpt-4o-mini",
        messages=[{"role": "user", "content": prompt}],
        temperature=0.3
    )
    return response.choices[0].message.content.strip()

# Exemples
reformulate_with_llm("trucs pour AD")
# → "Techniques de sécurisation d'Active Directory"

reformulate_with_llm("erreur quand j'installe")
# → "messages d'erreur lors de l'installation de logiciels"
```

Pipeline complet : En production, combinez ces techniques en cascade : correction orthographique → expansion acronymes → reformulation LLM (si nécessaire). Cela améliore le recall de 15-25% sur les requêtes mal formulées.

Recherche hybride : dense + sparse

Limitations de la recherche purement sémantique

Bien que puissante, la recherche vectorielle pure présente des faiblesses fondamentales que les approches hybrides résolvent. Comprendre ces limitations est crucial pour concevoir un système robuste.

Problèmes de la Recherche Dense Seule

1. Mauvaise Performance sur les Correspondances Exactes

- **Problème** : Les embeddings ne capturent pas bien les identifiants exacts (codes, références, numéros)
- **Exemple** : Recherche "CVE-2024-1234" peut renvoyer d'autres CVE similaires mais pas celui-là
- **Raison** : Les numéros/codes sont embeddés sémantiquement, pas lexicalement

2. Sensibilité aux Variantes Lexicales

- **Problème** : Variantes orthographiques ("email" vs "e-mail") ou abréviations
- **Exemple** : "COVID-19" vs "Covid" vs "coronavirus" → embeddings différents
- **Impact** : Recall incomplet sur des termes pourtant identiques

3. Difficulté avec les Termes Rares

- **Problème** : Les mots techniques/rares sont mal représentés dans l'espace d'embedding
- **Exemple** : "Kerberoasting" (attaque AD spécifique) embeddé proche de "Kerberos" général
- **Raison** : Peu d'occurrences dans les données d'entraînement du modèle

4. Absence de Scoring Lexical

- **Problème** : Pas de prise en compte de la fréquence des termes (TF-IDF)
- **Exemple** : Un document mentionnant 10x "cybersecurité" pas nécessairement mieux classé
- **Solution** : BM25 pondère les occurrences multiples

Benchmark Comparatif : Dense vs Sparse vs Hybrid

Dataset	Dense seul (NDCG@10)	BM25 seul	Hybrid	Gain hybrid
MS MARCO	0.78	0.72	0.82	+5%
Natural Questions	0.81	0.68	0.84	+4%
Technical Docs	0.74	0.79	0.86	+9%
Code Search	0.69	0.82	0.88	+7%

Observation clé : Sur les corpus techniques (docs, code), BM25 surpasse souvent le dense seul, car les correspondances lexicales exactes sont critiques. L'hybride combine le meilleur des deux mondes.

Combiner recherche dense (embeddings) et sparse (BM25)

La recherche hybride exécute en parallèle une recherche dense (similarité vectorielle) et sparse (BM25), puis fusionne les résultats. Voici une implémentation complète avec Elasticsearch + Qdrant.

Architecture Hybride avec Elasticsearch et Qdrant

```
import asyncio
from elasticsearch import Elasticsearch
from qdrant_client import QdrantClient
from qdrant_client.models import Distance, VectorParams, PointStruct
from sentence_transformers import SentenceTransformer
from typing import List, Dict, Tuple
import numpy as np

class HybridSearchEngine:
    def __init__(self, es_url: str = "http://localhost:9200",
                 qdrant_url: str = "http://localhost:6333"):
        # Clients
        self.es = Elasticsearch(es_url)
        self.qdrant = QdrantClient(url=qdrant_url)
        self.embedder = SentenceTransformer("BAAI/bge-large-en-v1.5")

        # Noms des collections
        self.es_index = "documents"
        self.qdrant_collection = "documents_vectors"

    def index_documents(self, documents: List[Dict[str, str]]):
        """
        Indexe documents dans ES (BM25) et Qdrant (vectors).

        Args:
            documents: [{"id": "1", "text": "...", "metadata": {...}}, ...]
        """
        # 1. Indexation Elasticsearch (BM25)
        for doc in documents:
            self.es.index(
                index=self.es_index,
                id=doc["id"],
                document={"text": doc["text"], **doc.get("metadata", {})}
            )

        # 2. Génération embeddings
        texts = [doc["text"] for doc in documents]
        embeddings = self.embedder.encode(texts, normalize_embeddings=True)

        # 3. Indexation Qdrant (vectors)
        points = [
            PointStruct(
                id=int(doc["id"]),
                vector=embeddings[i].tolist(),
                payload={"text": doc["text"], **doc.get("metadata", {})}
            )
            for i, doc in enumerate(documents)
        ]
        self.qdrant.upsert(collection_name=self.qdrant_collection, points=points)

        print(f"Indexé {len(documents)} documents (ES + Qdrant)")

    async def search_bm25(self, query: str, k: int = 100) -> List[Tuple[str, float]]:
        """Recherche BM25 avec Elasticsearch."""
        result = self.es.search(
            index=self.es_index,
            body={
                "query": {"match": {"text": query}},
                "size": k
            }
        )
```

```
        return [(hit["_id"], hit["_score"]) for hit in result["hits"]["hits"]]

    async def search_vector(self, query: str, k: int = 100) -> List[Tuple[str, float]]:
        """Recherche vectorielle avec Qdrant."""
        query_vector = self.embedder.encode(query, normalize_embeddings=True)
        results = self.qdrant.search(
            collection_name=self.qdrant_collection,
            query_vector=query_vector.tolist(),
            limit=k
        )
        return [(str(hit.id), hit.score) for hit in results]

    async def hybrid_search(self, query: str, k: int = 10,
                            alpha: float = 0.5) -> List[Dict]:
        """
        Recherche hybride avec fusion par pondération.

        Args:
            query: Requête utilisateur
            k: Nombre de résultats finaux
            alpha: Pondération (0=BM25 pur, 1=vector pur, 0.5=équilibré)

        Returns:
            Liste de {doc_id, score, text} triée par score hybride
        """
        # Exécuter les deux recherches en parallèle
        bm25_task = self.search_bm25(query, k=100)
        vector_task = self.search_vector(query, k=100)
        bm25_results, vector_results = await asyncio.gather(bm25_task, vector_task)

        # Normaliser les scores (min-max scaling)
        def normalize_scores(results: List[Tuple[str, float]]) -> Dict[str, float]:
            if not results:
                return {}
            scores = [s for _, s in results]
            min_s, max_s = min(scores), max(scores)
            if max_s == min_s:
                return {doc_id: 1.0 for doc_id, _ in results}
            return {
                doc_id: (score - min_s) / (max_s - min_s)
                for doc_id, score in results
            }

        bm25_scores = normalize_scores(bm25_results)
        vector_scores = normalize_scores(vector_results)

        # Fusion des scores
        all_doc_ids = set(bm25_scores.keys()) | set(vector_scores.keys())
        hybrid_scores = {}
        for doc_id in all_doc_ids:
            bm25_s = bm25_scores.get(doc_id, 0.0)
            vector_s = vector_scores.get(doc_id, 0.0)
            hybrid_scores[doc_id] = (1 - alpha) * bm25_s + alpha * vector_s

        # Trier et retourner top-k
        sorted_docs = sorted(hybrid_scores.items(), key=lambda x: x[1], reverse=True)[:k]

        # Récupérer les textes depuis Qdrant
        results = []
        for doc_id, score in sorted_docs:
            doc = self.qdrant.retrieve(
                collection_name=self.qdrant_collection,
```

```
                ids=[int(doc_id)]
            )[0]
            results.append({
                "doc_id": doc_id,
                "score": score,
                "text": doc.payload["text"]
            })

        return results

# Exemple d'utilisation
async def main():
    engine = HybridSearchEngine()

    # Indexation
    docs = [
        {"id": "1", "text": "Les embeddings convertissent le texte en vecteurs."},
        {"id": "2", "text": "BM25 est un algorithme de ranking probabiliste."},
        {"id": "3", "text": "La recherche hybride combine dense et sparse retrieval."}
    ]
    engine.index_documents(docs)

    # Recherche hybride
    results = await engine.hybrid_search(
        "comment combiner recherche sémantique et lexicale ?",
        k=3,
        alpha=0.5
    )

    for i, res in enumerate(results, 1):
        print(f"{i}. [Score: {res['score']:.3f}] {res['text']}")

if __name__ == "__main__":
    asyncio.run(main())
```

Cette implémentation exécute les deux recherches en **parallèle** (asyncio) pour minimiser la latence totale : ~70-120ms au lieu de 140-200ms en séquentiel.

Fusion des scores (RRF, weighted fusion)

Il existe plusieurs stratégies pour fusionner les résultats de recherches multiples. Chaque méthode a ses avantages selon le contexte.

1. Reciprocal Rank Fusion (RRF)
Méthode simple et robuste qui pondère par les **rangs** plutôt que les scores bruts. Idéal quand les échelles de scores sont incomparables.

```
def reciprocal_rank_fusion(results_list: List[List[Tuple[str, float]]],
                           k: int = 60) -> List[Tuple[str, float]]:
    """
    Fusion RRF de plusieurs listes de résultats.

    Args:
        results_list: [[('doc1', score), ...], [('doc2', score), ...], ...]
        k: Paramètre de pondération (60 est standard)

    Returns:
        Liste triée par score RRF
    """
    rrf_scores = {}

    for results in results_list:
        for rank, (doc_id, _) in enumerate(results, 1):
            if doc_id not in rrf_scores:
                rrf_scores[doc_id] = 0.0
            # Score RRF : somme des 1/(k+rank)
            rrf_scores[doc_id] += 1.0 / (k + rank)

    # Trier par score décroissant
    sorted_results = sorted(rrf_scores.items(), key=lambda x: x[1], reverse=True)
    return sorted_results

# Exemple
bm25_results = [('doc1', 10.5), ('doc2', 8.3), ('doc3', 7.1)]
vector_results = [('doc2', 0.95), ('doc3', 0.89), ('doc1', 0.82)]

fused = reciprocal_rank_fusion([bm25_results, vector_results], k=60)
# Résultat : [('doc2', 0.0323), ('doc3', 0.0311), ('doc1', 0.0308)]
# doc2 est 1er car bien classé dans les deux (rank 2 et 1)
```

2. Weighted Score Fusion

Fusion par pondération des scores normalisés. Permet de privilégier une méthode sur l'autre.

```
def weighted_score_fusion(results_dict: Dict[str, List[Tuple[str, float]]],
                          weights: Dict[str, float]) -> List[Tuple[str, float]]:
    """
    Fusion pondérée avec normalisation min-max.

    Args:
        results_dict: {'bm25': [(doc, score), ...], 'vector': [(doc, score), ...]}
        weights: {'bm25': 0.3, 'vector': 0.7}  # Doit sommer à 1.0

    Returns:
        Liste triée par score fusionné
    """
    def normalize(results: List[Tuple[str, float]]) -> Dict[str, float]:
        if not results:
            return {}
        scores = [s for _, s in results]
        min_s, max_s = min(scores), max(scores)
        if max_s == min_s:
            return {doc: 1.0 for doc, _ in results}
        return {doc: (s - min_s) / (max_s - min_s) for doc, s in results}

    # Normaliser chaque source
    normalized = {source: normalize(results)
                  for source, results in results_dict.items()}

    # Fusionner avec pondération
    all_docs = set().union(*[set(d.keys()) for d in normalized.values()])
    fused_scores = {}
    for doc in all_docs:
        fused_scores[doc] = sum(
            weights[source] * normalized[source].get(doc, 0.0)
            for source in results_dict.keys()
        )

    return sorted(fused_scores.items(), key=lambda x: x[1], reverse=True)

# Exemple
results = {
    'bm25': [('doc1', 12.5), ('doc2', 9.8)],
    'vector': [('doc2', 0.92), ('doc3', 0.88)]
}
weights = {'bm25': 0.3, 'vector': 0.7}  # Privilégier le vectoriel

fused = weighted_score_fusion(results, weights)
# doc2 sera premier (présent dans les deux avec bon score vectoriel)
```

3. Distribution-Based Score Fusion (DBSF)

Méthode avancée qui normalise via la distribution statistique des scores (z-score).

```
import numpy as np

def dbsf_fusion(results_dict: Dict[str, List[Tuple[str, float]]],
                weights: Dict[str, float]) -> List[Tuple[str, float]]:
    """
    Fusion via z-score normalization.
    """
    def zscore_normalize(results: List[Tuple[str, float]]) -> Dict[str, float]:
        scores = np.array([s for _, s in results])
        mean, std = scores.mean(), scores.std()
        if std == 0:
            return {doc: 0.0 for doc, _ in results}
        return {doc: (s - mean) / std for doc, s in results}

    # Normaliser chaque source
    normalized = {source: zscore_normalize(results)
                  for source, results in results_dict.items()}

    # Fusionner
    all_docs = set().union(*[set(d.keys()) for d in normalized.values()])
    fused_scores = {}
    for doc in all_docs:
        fused_scores[doc] = sum(
            weights[source] * normalized[source].get(doc, 0.0)
            for source in results_dict.keys()
        )

    return sorted(fused_scores.items(), key=lambda x: x[1], reverse=True)
```

Comparaison des Méthodes

Méthode	Avantages	Inconvénients	Quand utiliser
RRF	Simple, robuste, pas de normalisation	Ignore magnitude des scores	Scores incomparables entre sources
Weighted	Contrôle précis, interprétable	Nécessite normalisation	Sources avec échelles similaires
DBSF	Statistiquement rigoureux	Plus complexe, nécessite volume	Large corpus (>10K docs)

Recommandation Production

RRF avec k=60 est le choix par défaut : simple, robuste, pas de paramètre à tuner. Utilisez weighted fusion si vous avez des données de validation pour optimiser les poids (ex: 30% BM25, 70% vector).

Implémentation pratique

Voici une implémentation production-ready avec gestion d'erreurs, métriques et caching.

```
import time
from functools import lru_cache
from typing import List, Dict, Optional
import hashlib
import logging

logger = logging.getLogger(__name__)

class ProductionHybridSearch:
    def __init__(self, es_client, qdrant_client, embedder,
                 cache_size: int = 1000, cache_ttl: int = 3600):
        self.es = es_client
        self.qdrant = qdrant_client
        self.embedder = embedder
        self.cache_ttl = cache_ttl

        # Métriques
        self.metrics = {
            'total_queries': 0,
            'cache_hits': 0,
            'avg_latency_ms': 0,
            'errors': 0
        }

    def _cache_key(self, query: str, k: int, alpha: float) -> str:
        """Génère clé de cache."""
        key_str = f"{query}|{k}|{alpha}"
        return hashlib.md5(key_str.encode()).hexdigest()

    @lru_cache(maxsize=1000)
    def _get_query_embedding(self, query: str) -> tuple:
        """Cache les embeddings de requêtes."""
        emb = self.embedder.encode(query, normalize_embeddings=True)
        return tuple(emb)  # Convertir en tuple pour hashable

    def search(self, query: str, k: int = 10, alpha: float = 0.5,
               filters: Optional[Dict] = None) -> Dict:
        """
        Recherche hybride avec métriques et gestion d'erreurs.

        Returns:
            {
                'results': [...],
                'metadata': {
                    'latency_ms': float,
                    'num_results': int,
                    'from_cache': bool
                }
            }
        """
        start_time = time.time()
        self.metrics['total_queries'] += 1

        try:
            # Recherches parallèles avec timeout
            bm25_results = self._search_bm25(query, k=100, filters=filters)
            vector_emb = self._get_query_embedding(query)
            vector_results = self._search_vector(list(vector_emb), k=100, filters=filters)

            # Fusion RRF
            fused = self._rrf_fusion([bm25_results, vector_results])
```

```
            # Top-k final
            final_results = fused[:k]

            # Métriques
            latency_ms = (time.time() - start_time) * 1000
            self._update_metrics(latency_ms)

            return {
                'results': final_results,
                'metadata': {
                    'latency_ms': round(latency_ms, 2),
                    'num_results': len(final_results),
                    'from_cache': False,
                    'fusion_method': 'RRF',
                    'alpha': alpha
                }
            }

        except Exception as e:
            self.metrics['errors'] += 1
            logger.error(f"Erreur recherche hybride: {e}")
            # Fallback : recherche vectorielle seule
            return self._fallback_search(query, k)

    def _search_bm25(self, query: str, k: int, filters: Optional[Dict]) -> List:
        """Recherche BM25."""
        query_body = {"query": {"match": {"text": query}}, "size": k}
        if filters:
            query_body["query"] = {
                "bool": {
                    "must": query_body["query"],
                    "filter": [{
                        "term": {field: value}
                    } for field, value in filters.items()]
                }
            }

        result = self.es.search(index="documents", body=query_body, timeout="5s")
        return [(hit["_id"], hit["_score"]) for hit in result["hits"]["hits"]]

    def _search_vector(self, query_vector: List[float], k: int,
                       filters: Optional[Dict]) -> List:
        """Recherche vectorielle."""
        search_params = {
            "collection_name": "documents",
            "query_vector": query_vector,
            "limit": k
        }
        if filters:
            search_params["query_filter"] = {
                "must": [
                    {"key": field, "match": {"value": value}}
                    for field, value in filters.items()
                ]
            }

        results = self.qdrant.search(**search_params)
        return [(str(hit.id), hit.score) for hit in results]

    def _rrf_fusion(self, results_list: List[List], k: int = 60) -> List:
        """Reciprocal Rank Fusion."""
```

```
        rrf_scores = {}
        for results in results_list:
            for rank, (doc_id, _) in enumerate(results, 1):
                if doc_id not in rrf_scores:
                    rrf_scores[doc_id] = 0.0
                rrf_scores[doc_id] += 1.0 / (k + rank)
        return sorted(rrf_scores.items(), key=lambda x: x[1], reverse=True)

    def _fallback_search(self, query: str, k: int) -> Dict:
        """Recherche de secours en cas d'erreur."""
        logger.warning("Utilisation du fallback (vector only)")
        vector_emb = self._get_query_embedding(query)
        results = self._search_vector(list(vector_emb), k, filters=None)
        return {
            'results': results[:k],
            'metadata': {'fallback': True, 'method': 'vector_only'}
        }

    def _update_metrics(self, latency_ms: float):
        """Met à jour les métriques."""
        n = self.metrics['total_queries']
        current_avg = self.metrics['avg_latency_ms']
        self.metrics['avg_latency_ms'] = (current_avg * (n-1) + latency_ms) / n

    def get_metrics(self) -> Dict:
        """Retourne les métriques."""
        return {
            **self.metrics,
            'cache_hit_rate': self.metrics['cache_hits'] / max(1,
self.metrics['total_queries']),
            'error_rate': self.metrics['errors'] / max(1, self.metrics['total_queries'])
        }
```

Cas où l'hybride est supérieur

L'approche hybride montre des gains significatifs dans des scénarios spécifiques. Voici quand l'implémenter :

✅ Hybride Fortement Recommandé

- **Documentation technique** : Codes, références, identifiants exacts critiques
- **Corpus multilingue** : BM25 capture mots-clés multilingues invariants
- **Recherche légale/médicale** : Précision lexicale + contexte sémantique
- **E-commerce** : Noms produits exacts + recherche sémantique attributs
- **Code search** : Noms fonctions/variables + similarité logique

⚠️ Hybride Optionnel

- **Contenu conversationnel** : Dense seul souvent suffisant (forums, chats)
- **Corpus homogène** : Peu de variation lexicale, embeddings performants seuls
- **Contrainte latence stricte** : Hybride ajoute 50-100ms

❌ Hybride Non Recommandé

- **Corpus très petit** : <1000 docs, overhead pas justifié
- **Recherche image/audio** : Pas de représentation lexicale pertinente

- **Ressources limitées** : Maintenir 2 systèmes coûteux

Benchmark Comparatif : Dense vs Hybride par Domaine

Domaine	Dense NDCG@10	Hybrid NDCG@10	Gain	Verdict
Documentation technique	0.74	0.86	+16%	⭐ Critique
Code source	0.69	0.88	+28%	⭐ Critique
E-commerce	0.81	0.89	+10%	✅ Recommandé
Knowledge base entreprise	0.78	0.84	+8%	✅ Recommandé
Articles de blog	0.83	0.86	+4%	⚠️ Optionnel
Conversations/chats	0.87	0.88	+1%	❌ Pas nécessaire

Filtrage et métadonnées intelligentes

Pre-filtering vs post-filtering

Le choix entre **pre-filtering** (filtrer avant la recherche vectorielle) et **post-filtering** (filtrer après) a un impact majeur sur les performances et la qualité des résultats.

Pre-Filtering (Filtre en Amont)

- **Méthode** : La base vectorielle ne cherche que parmi les documents matchant les filtres
- **Avantage** : Recherche plus rapide (moins de vecteurs à comparer)
- **Limite** : Peut manquer de résultats si le filtre est trop restrictif
- **Implémentation** : Utilisé par Qdrant, Pinecone, Weaviate

```
# Exemple avec Qdrant
from qdrant_client.models import Filter, FieldCondition, MatchValue

# Pre-filtering : seuls les docs de 2024 sont cherchés
results = qdrant_client.search(
    collection_name="documents",
    query_vector=query_embedding,
    query_filter=Filter(
        must=[
            FieldCondition(
                key="year",
                match=MatchValue(value=2024)
            ),
            FieldCondition(
                key="category",
                match=MatchValue(value="security")
            )
        ]
    ),
    limit=10
)
```

Post-Filtering (Filtre en Aval)

- **Méthode** : Recherche sur tout le corpus, puis filtre les résultats
- **Avantage** : Garantit de toujours obtenir k résultats (si existants)
- **Limite** : Plus lent (doit chercher plus de candidats)
- **Implémentation** : Filtrage applicatif post-recherche

```
# Post-filtering : cherche 100, filtre, garde top-10
raw_results = qdrant_client.search(
    collection_name="documents",
    query_vector=query_embedding,
    limit=100  # Chercher plus large
)

# Filtrer en Python
filtered = [
    r for r in raw_results
    if r.payload.get("year") == 2024
    and r.payload.get("category") == "security"
][:10]  # Garder top-10
```

Stratégie Recommandée

Critère	Pre-Filtering	Post-Filtering
Sélectivité faible (>30% docs)	✅ Optimal	❌ Inefficace
Sélectivité élevée (<5% docs)	⚠️ Risque manque résultats	✅ Plus sûr
Latence critique	✅ Plus rapide	❌ Plus lent
Qualité garantie	❌ Peut manquer top-k	✅ Toujours k résultats

Filtres temporels et géographiques

Les filtres temporels et géographiques sont parmi les plus courants en production. Voici comment les implémenter efficacement.

Filtres Temporels

```
from datetime import datetime, timedelta
from qdrant_client.models import Filter, FieldCondition, Range

# Recherche sur les 30 derniers jours
thirty_days_ago = int((datetime.now() - timedelta(days=30)).timestamp())

results = qdrant_client.search(
    collection_name="documents",
    query_vector=query_embedding,
    query_filter=Filter(
        must=[
            FieldCondition(
                key="timestamp",
                range=Range(
                    gte=thirty_days_ago  # Greater than or equal
                )
            )
        ]
    ),
    limit=10
)

# Filtres temporels courants
filters_exemples = {
    "Dernière semaine": Range(gte=now - 7*24*3600),
    "Ce mois": Range(gte=first_day_of_month, lte=last_day_of_month),
    "Année 2024": Range(gte=1704067200, lte=1735689599),  # timestamps
    "Avant 2020": Range(lte=1577836800)
}
```

Filtres Géographiques

```
from qdrant_client.models import Filter, FieldCondition, GeoRadius, GeoPoint

# Recherche dans un rayon de 10km autour de Paris
results = qdrant_client.search(
    collection_name="locations",
    query_vector=query_embedding,
    query_filter=Filter(
        must=[
            FieldCondition(
                key="location",
                geo_radius=GeoRadius(
                    center=GeoPoint(lat=48.8566, lon=2.3522),  # Paris
                    radius=10000  # 10km en mètres
                )
            )
        ]
    ),
    limit=10
)

# Filtre par polygone (zone géographique complexe)
from qdrant_client.models import GeoPolygon

paris_polygon = GeoPolygon(
    exterior=GeoLineString(
        points=[
            GeoPoint(lat=48.9, lon=2.2),
            GeoPoint(lat=48.9, lon=2.5),
            GeoPoint(lat=48.8, lon=2.5),
            GeoPoint(lat=48.8, lon=2.2),
            GeoPoint(lat=48.9, lon=2.2)  # Fermer le polygone
        ]
    )
)
```

Filtres catégoriels

Les filtres sur catégories, tags ou métadonnées structurées sont essentiels pour affiner les résultats.

```
# Filtres multiples avec logique booléenne
from qdrant_client.models import Filter, FieldCondition, MatchAny, MatchValue

results = qdrant_client.search(
    collection_name="documents",
    query_vector=query_embedding,
    query_filter=Filter(
        must=[  # ET logique
            FieldCondition(key="status", match=MatchValue(value="published")),
            FieldCondition(
                key="category",
                match=MatchAny(any=["security", "compliance"])  # OU logique
            )
        ],
        must_not=[  # SAUF
            FieldCondition(key="archived", match=MatchValue(value=True))
        ],
        should=[  # BOOST (optionnel)
            FieldCondition(key="featured", match=MatchValue(value=True))
        ]
    ),
    limit=10
)

# Exemple complexe : e-commerce
e_commerce_filter = Filter(
    must=[
        FieldCondition(key="in_stock", match=MatchValue(value=True)),
        FieldCondition(key="price", range=Range(gte=10, lte=100)),
        FieldCondition(key="brand", match=MatchAny(any=["Nike", "Adidas"]))
    ],
    must_not=[
        FieldCondition(key="condition", match=MatchValue(value="used"))
    ]
)
```

Filtres dynamiques basés sur le contexte

Les filtres dynamiques s'adaptent automatiquement au contexte utilisateur (historique, préférences, permissions).

```
class ContextualSearchEngine:
    def __init__(self, qdrant_client, embedder):
        self.client = qdrant_client
        self.embedder = embedder

    def search_with_user_context(self, query: str, user_id: str,
                                 k: int = 10) -> List[Dict]:
        """
        Recherche avec filtres dynamiques basés sur le profil utilisateur.
        """
        # 1. Récupérer le contexte utilisateur
        user_profile = self._get_user_profile(user_id)

        # 2. Construire les filtres dynamiquement
        filters = self._build_dynamic_filters(user_profile)

        # 3. Recherche vectorielle filtrée
        query_vector = self.embedder.encode(query)
        results = self.client.search(
            collection_name="documents",
            query_vector=query_vector,
            query_filter=filters,
            limit=k
        )

        return results

    def _build_dynamic_filters(self, user_profile: Dict) -> Filter:
        """
        Construit les filtres selon le profil utilisateur.
        """
        must_conditions = []

        # Filtre permissions (sécurité)
        must_conditions.append(
            FieldCondition(
                key="access_level",
                range=Range(lte=user_profile["clearance_level"])
            )
        )

        # Filtre département (visibilité)
        must_conditions.append(
            FieldCondition(
                key="department",
                match=MatchAny(any=user_profile["allowed_departments"])
            )
        )

        # Filtre langue préférée
        if user_profile.get("preferred_language"):
            must_conditions.append(
                FieldCondition(
                    key="language",
                    match=MatchValue(value=user_profile["preferred_language"])
                )
            )

        # Filtre fraîcheur (si utilisateur préfère contenu récent)
        if user_profile.get("prefer_recent"):
            thirty_days_ago = int((datetime.now() - timedelta(days=30)).timestamp())
```

```
            must_conditions.append(
                FieldCondition(
                    key="updated_at",
                    range=Range(gte=thirty_days_ago)
                )
            )

        return Filter(must=must_conditions)

    def _get_user_profile(self, user_id: str) -> Dict:
        """
        Récupère le profil depuis base de données ou cache.
        """
        # Simuler récupération
        return {
            "clearance_level": 3,
            "allowed_departments": ["engineering", "security"],
            "preferred_language": "fr",
            "prefer_recent": True
        }
```

Impact sur les performances

Les filtres ont un impact direct sur la latence et la qualité. Voici des benchmarks réels sur un corpus de 10M documents avec Qdrant :

Type de filtre	Sélectivité	Latence (pre-filter)	Latence (post-filter)	Recommandation
Aucun filtre	100%	45ms	45ms	Baseline
Catégorie unique	20%	38ms	120ms	✅ Pre-filter
Date range (30j)	8%	32ms	180ms	✅ Pre-filter
Multi-filtres (3+)	2%	65ms	250ms	⚠️ Hybride
Filtre très restrictif	0.1%	150ms*	200ms	❌ Post-filter

* Le pre-filter très restrictif doit chercher dans beaucoup plus de candidats pour trouver k résultats

Règle d'Or du Filtrage

- **Sélectivité >10%** : Pre-filtering optimal
- **Sélectivité 5-10%** : Pre-filter avec over-fetching (chercher 2-3x plus)
- **Sélectivité <5%** : Post-filtering ou approche hybride
- **Permissions critiques** : Toujours pre-filter (sécurité)

Personnalisation des résultats

User embeddings et profils utilisateurs

Les **user embeddings** capturent les préférences et intérêts d'un utilisateur sous forme vectorielle, permettant une personnalisation sémantique de la recherche.

Construction d'un User Embedding

```
import numpy as np
from collections import defaultdict
from datetime import datetime, timedelta

class UserEmbeddingBuilder:
    def __init__(self, embedder, decay_days: int = 90):
        self.embedder = embedder
        self.decay_days = decay_days

    def build_user_embedding(self, user_id: str,
                             user_history: List[Dict]) -> np.ndarray:
        """
        Construit un embedding utilisateur depuis son historique.

        Args:
            user_history: [{"doc_id": ..., "timestamp": ..., "interaction_type": ...}]

        Returns:
            Vecteur représentant les intérêts de l'utilisateur
        """
        embeddings_weighted = []
        now = datetime.now()

        for interaction in user_history:
            # Récupérer l'embedding du document
            doc_embedding = self._get_doc_embedding(interaction["doc_id"])

            # Calculer le poids selon le type d'interaction
            interaction_weight = self._get_interaction_weight(
                interaction["interaction_type"]
            )

            # Appliquer décroissance temporelle
            days_ago = (now - interaction["timestamp"]).days
            time_decay = np.exp(-days_ago / self.decay_days)

            # Poids final
            weight = interaction_weight * time_decay
            embeddings_weighted.append(doc_embedding * weight)

        if not embeddings_weighted:
            # Utilisateur nouveau : embedding neutre
            return np.zeros(self.embedder.get_sentence_embedding_dimension())

        # Moyenne pondérée des embeddings
        user_embedding = np.mean(embeddings_weighted, axis=0)
        # Normaliser
        user_embedding = user_embedding / np.linalg.norm(user_embedding)

        return user_embedding

    def _get_interaction_weight(self, interaction_type: str) -> float:
        """
        Pondération selon le type d'interaction.
        """
        weights = {
            "viewed": 1.0,
            "clicked": 2.0,
            "bookmarked": 3.0,
            "shared": 4.0,
            "downloaded": 5.0
```

```
        }
        return weights.get(interaction_type, 1.0)

    def personalized_search(self, query: str, user_embedding: np.ndarray,
                            alpha: float = 0.7) -> List:
        """
        Recherche personnalisée combinant query et profil utilisateur.

        Args:
            alpha: Pondération (1=query pur, 0=profil pur)
        """
        # Embedding de la requête
        query_embedding = self.embedder.encode(query, normalize_embeddings=True)

        # Fusion query + user profile
        personalized_embedding = (
            alpha * query_embedding +
            (1 - alpha) * user_embedding
        )
        personalized_embedding /= np.linalg.norm(personalized_embedding)

        # Recherche avec l'embedding personnalisé
        results = self.vector_db.search(
            query_vector=personalized_embedding.tolist(),
            limit=10
        )

        return results
```

Gains observés : La personnalisation via user embeddings améliore le CTR de 15-30% et la satisfaction utilisateur de 20-35% sur des systèmes de recommandation et recherche.

Historique de recherche et feedback

L'exploitation de l'historique de recherche et du feedback utilisateur permet d'améliorer continuellement la pertinence.

Système de Feedback Implicite et Explicite

```
from enum import Enum

class FeedbackType(Enum):
    IMPLICIT_CLICK = 1      # Utilisateur a cliqué
    IMPLICIT_TIME = 2       # Temps passé >30s
    EXPLICIT_THUMBS_UP = 3  # Pouce haut
    EXPLICIT_THUMBS_DOWN = 4 # Pouce bas
    EXPLICIT_REPORT = 5     # Signalé comme non pertinent

class FeedbackTracker:
    def __init__(self, db_client):
        self.db = db_client

    def track_feedback(self, user_id: str, query: str,
                       doc_id: str, feedback_type: FeedbackType,
                       rank_position: int):
        """
        Enregistre le feedback utilisateur.
        """
        feedback_entry = {
            "user_id": user_id,
            "query": query,
            "doc_id": doc_id,
            "feedback_type": feedback_type.name,
            "rank_position": rank_position,
            "timestamp": datetime.now(),
            "weight": self._get_feedback_weight(feedback_type, rank_position)
        }
        self.db.insert("feedback", feedback_entry)

    def _get_feedback_weight(self, feedback_type: FeedbackType,
                             rank_position: int) -> float:
        """
        Calcule le poids du feedback.
        """
        base_weights = {
            FeedbackType.IMPLICIT_CLICK: 1.0,
            FeedbackType.IMPLICIT_TIME: 1.5,
            FeedbackType.EXPLICIT_THUMBS_UP: 3.0,
            FeedbackType.EXPLICIT_THUMBS_DOWN: -2.0,
            FeedbackType.EXPLICIT_REPORT: -5.0
        }

        # Pondérer selon la position (clic en position 1 vaut plus que position 10)
        position_factor = 1.0 / np.log2(rank_position + 1)

        return base_weights[feedback_type] * position_factor

    def get_query_performance(self, query: str, days: int = 30) -> Dict:
        """
        Analyse les performances d'une requête.
        """
        feedbacks = self.db.query(
            "SELECT * FROM feedback WHERE query = ? AND timestamp > ?",
            (query, datetime.now() - timedelta(days=days))
        )

        return {
            "total_searches": len(feedbacks),
            "click_through_rate": self._compute_ctr(feedbacks),
            "avg_rank_clicked": self._compute_avg_rank(feedbacks),
```

```
            "satisfaction_score": self._compute_satisfaction(feedbacks)
        }

    def _compute_ctr(self, feedbacks: List[Dict]) -> float:
        """Taux de clic."""
        clicks = [f for f in feedbacks if "CLICK" in f["feedback_type"]]
        return len(clicks) / max(1, len(feedbacks))

    def _compute_avg_rank(self, feedbacks: List[Dict]) -> float:
        """Position moyenne des clics."""
        clicks = [f for f in feedbacks if "CLICK" in f["feedback_type"]]
        if not clicks:
            return 0.0
        return np.mean([f["rank_position"] for f in clicks])

    def _compute_satisfaction(self, feedbacks: List[Dict]) -> float:
        """Score de satisfaction global."""
        if not feedbacks:
            return 0.0
        total_weight = sum(f["weight"] for f in feedbacks)
        return total_weight / len(feedbacks)
```

Contextual search

La recherche contextuelle adapte les résultats selon le contexte de la session : navigation précédente, heure, appareil, etc.

```
class ContextualSearchEngine:
    def search_with_context(self, query: str, context: Dict) -> List[Dict]:
        """
        Recherche avec prise en compte du contexte.

        Args:
            context: {
                "session_history": [...],  # Pages vues dans la session
                "time_of_day": "morning",
                "device": "mobile",
                "location": "Paris",
                "previous_query": "..."
            }
        """
        # 1. Enrichir la requête avec le contexte
        enriched_query = self._enrich_query(query, context)

        # 2. Ajuster les filtres selon le contexte
        filters = self._build_contextual_filters(context)

        # 3. Recherche standard
        results = self.search(enriched_query, filters=filters)

        # 4. Réordonnancer selon le contexte
        reranked = self._contextual_rerank(results, context)

        return reranked

    def _enrich_query(self, query: str, context: Dict) -> str:
        """
        Enrichit la requête avec le contexte de session.
        """
        # Si requête liée à la précédente
        if context.get("previous_query") and self._is_followup(query):
            return f"{context['previous_query']} {query}"

        # Si navigation précédente suggère un thème
        if context.get("session_history"):
            theme = self._extract_theme(context["session_history"])
            if theme:
                return f"{query} {theme}"

        return query

    def _build_contextual_filters(self, context: Dict) -> Filter:
        """
        Construit des filtres adaptés au contexte.
        """
        filters = []

        # Filtres selon l'appareil
        if context.get("device") == "mobile":
            # Privilégier les contenus courts sur mobile
            filters.append(
                FieldCondition(key="content_length", range=Range(lte=2000))
            )

        # Filtres selon l'heure
        if context.get("time_of_day") == "morning":
            # Le matin : actualités récentes
            filters.append(
```

```
                FieldCondition(
                    key="published_at",
                    range=Range(gte=int((datetime.now() - timedelta(days=1)).timestamp()))
                )
            )

        # Filtres géographiques
        if context.get("location"):
            filters.append(
                FieldCondition(key="region", match=MatchValue(value=context["location"]))
            )

        return Filter(must=filters) if filters else None

    def _contextual_rerank(self, results: List, context: Dict) -> List:
        """
        Réordonne selon le contexte de session.
        """
        if not context.get("session_history"):
            return results

        # Calculer similarité avec documents vus
        session_docs = context["session_history"]
        for result in results:
            # Boost si lié aux docs déjà vus
            similarity_boost = self._compute_session_similarity(
                result["doc_id"], session_docs
            )
            result["score"] *= (1 + 0.2 * similarity_boost)  # Boost jusqu'à 20%

        return sorted(results, key=lambda x: x["score"], reverse=True)
```

A/B testing et apprentissage continu

L'A/B testing permet d'optimiser progressivement les paramètres du système de recherche en mesurant l'impact réel sur les utilisateurs.

Mise en pratique

Framework d'A/B Testing pour Recherche

```
import hashlib
from typing import Dict, List
import random

class SearchABTestFramework:
    def __init__(self):
        self.experiments = {}
        self.metrics_tracker = MetricsTracker()

    def register_experiment(self, experiment_id: str,
                            variants: Dict[str, Dict]):
        """
        Enregistre une expérience A/B.

        Args:
            experiment_id: Identifiant unique
            variants: {
                "control": {"reranking": False, "k": 50, "alpha": 0.5},
                "variant_a": {"reranking": True, "k": 100, "alpha": 0.5},
                "variant_b": {"reranking": True, "k": 50, "alpha": 0.7}
            }
        """
        self.experiments[experiment_id] = {
            "variants": variants,
            "traffic_split": self._equal_split(len(variants)),
            "start_date": datetime.now(),
            "status": "active"
        }

    def assign_variant(self, user_id: str, experiment_id: str) -> str:
        """
        Assigne un utilisateur à une variante (stable hash-based).
        """
        if experiment_id not in self.experiments:
            return "control"

        # Hash utilisateur + expérience pour assignation stable
        hash_input = f"{user_id}:{experiment_id}"
        hash_value = int(hashlib.md5(hash_input.encode()).hexdigest(), 16)

        variants = list(self.experiments[experiment_id]["variants"].keys())
        variant_index = hash_value % len(variants)

        return variants[variant_index]

    def search_with_experiment(self, query: str, user_id: str,
                               experiment_id: str = "reranking_test"):
        """
        Exécute une recherche selon la variante assignée.
        """
        # Assigner variante
        variant = self.assign_variant(user_id, experiment_id)
        config = self.experiments[experiment_id]["variants"][variant]

        # Exécuter recherche avec config spécifique
        start_time = time.time()
        results = self._execute_search(query, config)
        latency = time.time() - start_time

        # Tracker l'impression
        self.metrics_tracker.track_impression(
```

```
            experiment_id=experiment_id,
            variant=variant,
            user_id=user_id,
            query=query,
            results=results,
            latency=latency
        )

        return results

    def analyze_experiment(self, experiment_id: str,
                           min_samples: int = 1000) -> Dict:
        """
        Analyse statistique des résultats.
        """
        metrics = self.metrics_tracker.get_metrics(experiment_id)

        # Calculer pour chaque variante
        analysis = {}
        for variant, data in metrics.items():
            if len(data["impressions"]) < min_samples:
                continue

            analysis[variant] = {
                "impressions": len(data["impressions"]),
                "ctr": np.mean(data["clicks"]),
                "avg_latency": np.mean(data["latencies"]),
                "satisfaction": np.mean(data["satisfaction_scores"]),
                "revenue_per_search": np.mean(data["conversions"]) * avg_order_value
            }

        # Test statistique (Student's t-test)
        if "control" in analysis and len(analysis) > 1:
            for variant in analysis:
                if variant != "control":
                    p_value = self._compute_significance(
                        analysis["control"],
                        analysis[variant]
                    )
                    analysis[variant]["p_value"] = p_value
                    analysis[variant]["significant"] = p_value < 0.05

        return analysis

    def _execute_search(self, query: str, config: Dict) -> List:
        """Exécute recherche avec configuration spécifique."""
        # Recherche de base
        results = self.base_search(query, k=config["k"], alpha=config["alpha"])

        # Reranking si activé
        if config.get("reranking", False):
            results = self.reranker.rerank(query, results)

        return results

# Exemple d'utilisation
ab_framework = SearchABTestFramework()

# Expérience : tester l'impact du reranking
ab_framework.register_experiment(
    "reranking_impact_2025",
    variants={
```

```
        "control": {"reranking": False, "k": 50},
        "reranking_enabled": {"reranking": True, "k": 100}
    }
)

# Après 2 semaines et 10K recherches
results = ab_framework.analyze_experiment("reranking_impact_2025")
# Résultats typiques :
# control: CTR 12%, latency 85ms, satisfaction 3.2/5
# reranking: CTR 15.5% (+29%), latency 320ms, satisfaction 3.8/5, p=0.002 (significatif!)
```

Métriques Clés à Tracker en A/B Testing

Mise en oeuvre pratique

- **CTR** : Taux de clic sur les résultats
- **Time to click** : Temps avant le premier clic
- **Satisfaction** : Feedback utilisateur (thumbs up/down)
- **Session success rate** : % sessions avec conversion
- **Latence p95** : 95e percentile de latence
- **Zero-result rate** : % requêtes sans résultats

Métriques et évaluation de qualité

Métriques classiques (Precision, Recall, F1)

Les métriques de base de l'information retrieval restent fondamentales pour évaluer la qualité d'un système de recherche.

Définitions et Formules
Precision (Précision)

Proportion de documents pertinents parmi ceux retournés :

Precision@k = (Nombre de docs pertinents dans top-k) / k

Exemple : Si sur 10 résultats, 7 sont pertinents → Precision@10 = 0.70 Pour approfondir, consultez Multimodal RAG 2026 : Texte, Image, Audio.

Recall (Rappel)

Proportion de documents pertinents retrouvés parmi tous ceux existants :

Recall@k = (Docs pertinents dans top-k) / (Total docs pertinents)

Exemple : Si 7 docs sur 20 pertinents totaux → Recall@10 = 0.35

F1-Score

Moyenne harmonique entre Precision et Recall :

F1 = 2 × (Precision × Recall) / (Precision + Recall)

Exemple : Precision=0.70, Recall=0.35 → F1 = 0.467

Implémentation Python

```
import numpy as np
from typing import List, Set

def precision_at_k(retrieved: List[str], relevant: Set[str], k: int) -> float:
    """
    Calcule Precision@k.

    Args:
        retrieved: Liste des doc_ids retournés (ordonnés par score)
        relevant: Ensemble des doc_ids pertinents (ground truth)
        k: Nombre de résultats à considérer

    Returns:
        Precision@k entre 0 et 1
    """
    retrieved_at_k = set(retrieved[:k])
    num_relevant_retrieved = len(retrieved_at_k & relevant)
    return num_relevant_retrieved / k if k > 0 else 0.0

def recall_at_k(retrieved: List[str], relevant: Set[str], k: int) -> float:
    """
    Calcule Recall@k.
    """
    retrieved_at_k = set(retrieved[:k])
    num_relevant_retrieved = len(retrieved_at_k & relevant)
    total_relevant = len(relevant)
    return num_relevant_retrieved / total_relevant if total_relevant > 0 else 0.0

def f1_at_k(retrieved: List[str], relevant: Set[str], k: int) -> float:
    """
    Calcule F1@k.
    """
    prec = precision_at_k(retrieved, relevant, k)
    rec = recall_at_k(retrieved, relevant, k)
    if prec + rec == 0:
        return 0.0
    return 2 * (prec * rec) / (prec + rec)

# Exemple
retrieved_docs = ["doc1", "doc2", "doc3", "doc4", "doc5", "doc6", "doc7", "doc8", "doc9",
"doc10"]
relevant_docs = {"doc1", "doc3", "doc5", "doc8", "doc12", "doc15", "doc20"}  # 7
pertinents totaux

print(f"Precision@10: {precision_at_k(retrieved_docs, relevant_docs, 10):.3f}")  # 0.400
print(f"Recall@10: {recall_at_k(retrieved_docs, relevant_docs, 10):.3f}")     # 0.571
print(f"F1@10: {f1_at_k(retrieved_docs, relevant_docs, 10):.3f}")             # 0.471
```

MAP (Mean Average Precision)

MAP est une métrique plus complexe qui prend en compte l'**ordre des résultats**. Elle calcule la moyenne des précisions à chaque position où un document pertinent apparaît.

Formule et Calcul

AP = (1/|Relevant|) × Σ (Precision@k × rel(k))

où rel(k) = 1 si le doc en position k est pertinent, 0 sinon

MAP = Moyenne des AP sur toutes les requêtes

```
def average_precision(retrieved: List[str], relevant: Set[str]) -> float:
    """
    Calcule Average Precision pour une requête.
    """
    if not relevant:
        return 0.0

    precision_sum = 0.0
    num_relevant_seen = 0

    for k, doc_id in enumerate(retrieved, 1):
        if doc_id in relevant:
            num_relevant_seen += 1
            # Precision à cette position
            precision_at_k = num_relevant_seen / k
            precision_sum += precision_at_k

    return precision_sum / len(relevant)

def mean_average_precision(all_queries: List[Dict]) -> float:
    """
    Calcule MAP sur plusieurs requêtes.

    Args:
        all_queries: [{"retrieved": [...], "relevant": {...}}, ...]
    """
    aps = []
    for query_results in all_queries:
        ap = average_precision(
            query_results["retrieved"],
            query_results["relevant"]
        )
        aps.append(ap)

    return np.mean(aps) if aps else 0.0

# Exemple
retrieved = ["doc2", "doc1", "doc4", "doc3", "doc5"]  # doc1, doc3, doc5 pertinents
relevant = {"doc1", "doc3", "doc5"}

# Calcul manuel :
# Position 2 (doc1 pertinent) : P@2 = 1/2 = 0.50
# Position 4 (doc3 pertinent) : P@4 = 2/4 = 0.50
# Position 5 (doc5 pertinent) : P@5 = 3/5 = 0.60
# AP = (0.50 + 0.50 + 0.60) / 3 = 0.533

print(f"AP: {average_precision(retrieved, relevant):.3f}")  # 0.533
```

NDCG (Normalized Discounted Cumulative Gain)

NDCG est la métrique de référence pour évaluer les systèmes de ranking. Elle tient compte des **niveaux de pertinence** (pas seulement binaire) et applique un **rabais logarithmique** aux positions basses.

Formule

DCG@k = Σ (rel_i / log2(i + 1))

pour i de 1 à k

NDCG@k = DCG@k / IDCG@k

où IDCG = DCG du classement idéal

```
import numpy as np

def dcg_at_k(relevances: List[float], k: int) -> float:
    """
    Calcule DCG@k.

    Args:
        relevances: Scores de pertinence (ex: [3, 2, 3, 0, 1, 2]) pour chaque position
        k: Nombre de positions à considérer

    Returns:
        DCG score
    """
    relevances_at_k = relevances[:k]
    gains = [(2**rel - 1) / np.log2(i + 2) for i, rel in enumerate(relevances_at_k)]
    return sum(gains)

def ndcg_at_k(retrieved: List[str], relevance_scores: Dict[str, float], k: int) -> float:
    """
    Calcule NDCG@k.

    Args:
        retrieved: Liste des doc_ids retournés (ordonnés)
        relevance_scores: {doc_id: score_pertinence} (0=non pertinent, 3=très pertinent)
        k: Position

    Returns:
        NDCG@k entre 0 et 1
    """
    # DCG réel
    actual_relevances = [relevance_scores.get(doc_id, 0.0) for doc_id in retrieved]
    dcg = dcg_at_k(actual_relevances, k)

    # IDCG (classement idéal)
    ideal_relevances = sorted(relevance_scores.values(), reverse=True)
    idcg = dcg_at_k(ideal_relevances, k)

    return dcg / idcg if idcg > 0 else 0.0

# Exemple
retrieved_docs = ["doc1", "doc2", "doc3", "doc4", "doc5"]
relevance = {
    "doc1": 3,  # Très pertinent
    "doc2": 1,  # Peu pertinent
    "doc3": 2,  # Pertinent
    "doc4": 0,  # Non pertinent
    "doc5": 3,  # Très pertinent
    "doc6": 3,  # Non retourné mais très pertinent
}

print(f"NDCG@5: {ndcg_at_k(retrieved_docs, relevance, 5):.3f}")  # ~0.850
# Un classement idéal serait : [doc1, doc5, doc6, doc3, doc2] (tous les 3 puis 2 puis 1)
```

MRR (Mean Reciprocal Rank)

MRR mesure à quelle position apparaît le **premier document pertinent**. Utile pour évaluer les systèmes de Q&A où une seule bonne réponse suffit.

RR = 1 / rank(premier_doc_pertinent)

MRR = Moyenne des RR sur toutes les requêtes

Exemple : Si le premier résultat pertinent est en position 3 → RR = 1/3 = 0.333

```
def reciprocal_rank(retrieved: List[str], relevant: Set[str]) -> float:
    """
    Calcule Reciprocal Rank pour une requête.
    """
    for rank, doc_id in enumerate(retrieved, 1):
        if doc_id in relevant:
            return 1.0 / rank
    return 0.0  # Aucun document pertinent trouvé

def mean_reciprocal_rank(all_queries: List[Dict]) -> float:
    """
    Calcule MRR sur plusieurs requêtes.
    """
    rrs = []
    for query_results in all_queries:
        rr = reciprocal_rank(
            query_results["retrieved"],
            query_results["relevant"]
        )
        rrs.append(rr)

    return np.mean(rrs) if rrs else 0.0

# Exemples
queries = [
    {"retrieved": ["d1", "d2", "d3"], "relevant": {"d2"}},  # RR = 1/2 = 0.5
    {"retrieved": ["d4", "d5", "d6"], "relevant": {"d4"}},  # RR = 1/1 = 1.0
    {"retrieved": ["d7", "d8", "d9"], "relevant": {"d9"}},  # RR = 1/3 = 0.333
]

print(f"MRR: {mean_reciprocal_rank(queries):.3f}")  # (0.5 + 1.0 + 0.333) / 3 = 0.611
```

Construire un dataset d'évaluation

Un dataset d'évaluation de qualité est crucial pour mesurer objectivement les améliorations. Voici comment en construire un.

Méthode 1 : Annotation Manuelle

```
import pandas as pd
from typing import List, Dict

class EvaluationDatasetBuilder:
    def __init__(self):
        self.annotations = []

    def sample_queries(self, query_logs: List[str], n: int = 100) -> List[str]:
        """
        Sélectionne n requêtes représentatives depuis les logs.
        """
        # Stratégies de sampling
        freq_dist = Counter(query_logs)

        # Mix de requêtes fréquentes et rares
        top_50 = [q for q, _ in freq_dist.most_common(50)]  # 50% populaires
        rare_50 = random.sample([q for q in freq_dist if freq_dist[q] <= 5], 50)  # 50%
rares

        return top_50 + rare_50

    def annotate_results(self, query: str, retrieved_docs: List[Dict]):
        """
        Interface d'annotation pour un annotateur humain.
        """
        print(f"\n=== Requête : {query} ===")
        annotations = {}

        for i, doc in enumerate(retrieved_docs[:20], 1):  # Top-20 seulement
            print(f"\n[{i}] {doc['title']}")
            print(f"    {doc['snippet'][:200]}...")
            score = input("    Pertinence (0=non, 1=peu, 2=moyen, 3=très) : ")
            annotations[doc['id']] = int(score)

        self.annotations.append({
            "query": query,
            "relevance_scores": annotations,
            "annotator": "annotator_1",
            "timestamp": datetime.now()
        })

    def export_dataset(self, filepath: str):
        """Exporte le dataset au format JSON."""
        with open(filepath, 'w') as f:
            json.dump(self.annotations, f, indent=2, default=str)

# Processus complet
builder = EvaluationDatasetBuilder()

# 1. Sélectionner 100 requêtes représentatives
query_logs = load_query_logs()  # Depuis base de données
sampled_queries = builder.sample_queries(query_logs, n=100)

# 2. Annoter (peut prendre 20-40h pour 100 requêtes x 20 docs)
for query in sampled_queries:
    retrieved = search_engine.search(query, k=20)
    builder.annotate_results(query, retrieved)

# 3. Exporter
```

```
builder.export_dataset("eval_dataset_v1.json")
```

Méthode 2 : Annotation Semi-Automatique avec LLM

```
def llm_assisted_annotation(query: str, doc: Dict) -> int:
    """
    Utilise un LLM pour pré-annoter (puis validation humaine).
    """
    prompt = f"""Tu es un expert en évaluation de pertinence.

Requête : {query}
Document : {doc['title']}
{doc['content'][:500]}

Évalue la pertinence sur cette échelle :
0 - Non pertinent (hors sujet)
1 - Peu pertinent (légèrement lié)
2 - Pertinent (répond partiellement)
3 - Très pertinent (répond complètement)

Réponds uniquement avec le chiffre."""

    response = openai.chat.completions.create(
        model="gpt-4o",
        messages=[{"role": "user", "content": prompt}],
        temperature=0.0
    )

    try:
        score = int(response.choices[0].message.content.strip())
        return max(0, min(3, score))  # Clamp entre 0-3
    except:
        return 1  # Score par défaut en cas d'erreur

# Coût typique : $0.01-0.03 par annotation avec GPT-4o
# Pour 100 queries x 20 docs = 2000 annotations ≈ $20-60
```

Monitoring en production

Le monitoring continu en production permet de détecter les régressions et d'identifier les opportunités d'amélioration.

Points d'attention

Dashboard de Métriques Temps-Réel

```
from prometheus_client import Counter, Histogram, Gauge
import time

# Métriques Prometheus
search_requests_total = Counter(
    'search_requests_total',
    'Nombre total de requêtes de recherche',
    ['method', 'status']
)

search_latency = Histogram(
    'search_latency_seconds',
    'Latence des recherches',
    ['method'],
    buckets=[0.01, 0.05, 0.1, 0.25, 0.5, 1.0, 2.5, 5.0]
)

search_quality_score = Gauge(
    'search_quality_score',
    'Score de qualité moyen (CTR)',
    ['time_window']
)

zero_results_rate = Gauge(
    'zero_results_rate',
    'Taux de requêtes sans résultats'
)

class MonitoredSearchEngine:
    def __init__(self, base_engine):
        self.engine = base_engine
        self.recent_ctr = []  # Sliding window CTR

    def search(self, query: str, user_id: str) -> Dict:
        """
        Recherche avec monitoring intégré.
        """
        start_time = time.time()

        try:
            # Exécuter recherche
            results = self.engine.search(query)

            # Latence
            latency = time.time() - start_time
            search_latency.labels(method='hybrid').observe(latency)

            # Compteurs
            status = 'success' if results else 'zero_results'
            search_requests_total.labels(method='hybrid', status=status).inc()

            # Taux zéro résultat
            if not results:
                self._update_zero_results_rate()

            # Logger pour analyse
            self._log_search(query, user_id, results, latency)

            return results

        except Exception as e:
```

```
            search_requests_total.labels(method='hybrid', status='error').inc()
            raise

    def track_click(self, query: str, doc_id: str, rank: int):
        """
        Tracker les clics pour calculer le CTR.
        """
        # Mise à jour CTR glissant (dernières 1000 recherches)
        self.recent_ctr.append(1)  # Clic = 1
        if len(self.recent_ctr) > 1000:
            self.recent_ctr.pop(0)

        ctr = np.mean(self.recent_ctr)
        search_quality_score.labels(time_window='1h').set(ctr)

    def _update_zero_results_rate(self):
        """Calcule le taux de zéro résultats sur la dernière heure."""
        # Récupérer depuis base métriques
        recent_searches = get_recent_searches(hours=1)
        zero_count = sum(1 for s in recent_searches if s['num_results'] == 0)
        rate = zero_count / len(recent_searches) if recent_searches else 0
        zero_results_rate.set(rate)

# Configuration Grafana Dashboard
# Panel 1 : Latence p50/p95/p99
# Panel 2 : QPS (queries per second)
# Panel 3 : Taux d'erreur
# Panel 4 : CTR temps-réel
# Panel 5 : Zero-results rate
# Panel 6 : Distribution des latences (heatmap)
```

Alertes Recommandées

- **Latence p95 > 1s** : Alerte investigation
- **Taux erreur > 1%** : Alerte critique
- **Zero-results > 15%** : Dégradation qualité
- **CTR baisse >20%** : Régression possible
- **QPS spike >3x normale** : Risque surcharge

Optimisations pour la production

Caching intelligent

Le caching est l'optimisation la plus efficace pour réduire la latence et les coûts. Une stratégie de cache bien conçue peut réduire la charge de 60-80%.

Stratégie Multi-Niveaux

```
import redis
import hashlib
import json
from functools import lru_cache
from typing import Optional, List, Dict

class MultiLevelCacheSystem:
    def __init__(self, redis_client: redis.Redis):
        self.redis = redis_client
        # L1 : Cache in-memory (LRU)
        # L2 : Cache Redis (distribué)

    @lru_cache(maxsize=1000)  # L1 Cache (in-process)
    def get_query_embedding(self, query: str) -> tuple:
        """
        Cache L1 : Embeddings de requêtes en mémoire.
        """
        embedding = self.embedder.encode(query)
        return tuple(embedding)  # Tuple pour hashable

    def search_with_cache(self, query: str, k: int = 10) -> Optional[List[Dict]]:
        """
        Recherche avec cache L1 (in-memory) et L2 (Redis).
        """
        # Générer clé de cache
        cache_key = self._generate_cache_key(query, k)

        # L1 : Vérifier cache in-memory (nanoseconds)
        if hasattr(self, '_l1_cache') and cache_key in self._l1_cache:
            return self._l1_cache[cache_key]

        # L2 : Vérifier Redis (1-5ms)
        cached = self.redis.get(cache_key)
        if cached:
            results = json.loads(cached)
            # Stocker en L1 pour accès futurs
            if not hasattr(self, '_l1_cache'):
                self._l1_cache = {}
            self._l1_cache[cache_key] = results
            return results

        # L3 : Exécuter recherche réelle (50-500ms)
        results = self._execute_search(query, k)

        # Stocker dans les caches
        self._store_in_caches(cache_key, results, ttl=3600)  # 1h TTL

        return results

    def _generate_cache_key(self, query: str, k: int) -> str:
        """
        Génère clé de cache stable.
        """
        # Normaliser la requête
        normalized = query.lower().strip()
        key_str = f"search:{normalized}:k={k}"
        return hashlib.sha256(key_str.encode()).hexdigest()

    def _store_in_caches(self, key: str, results: List[Dict], ttl: int):
        """
        Stocke dans L1 et L2.
```

```
        """
        # L1 in-memory
        if not hasattr(self, '_l1_cache'):
            self._l1_cache = {}
        self._l1_cache[key] = results

        # L2 Redis
        self.redis.setex(
            key,
            ttl,
            json.dumps(results, default=str)
        )

    def warm_cache(self, popular_queries: List[str]):
        """
        Pré-chauffe le cache avec les requêtes populaires.
        À exécuter pendant les heures creuses.
        """
        for query in popular_queries:
            self.search_with_cache(query)
        print(f"Cache préchauffé avec {len(popular_queries)} requêtes")

# Utilisation
cache_system = MultiLevelCacheSystem(redis_client)

# Préchauffage nocturne
top_1000_queries = get_popular_queries(days=7, limit=1000)
cache_system.warm_cache(top_1000_queries)

# Statistiques typiques :
# - L1 hit rate : 10-15% (requêtes répétées dans même processus)
# - L2 hit rate : 40-60% (requêtes populaires)
# - Total cache hit rate : 50-75%
# - Latence L1 hit : <1ms
# - Latence L2 hit : 2-5ms
# - Latence cache miss : 80-500ms
```

Cache Invalidation Intelligente

```
class SmartCacheInvalidation:
    def on_document_update(self, doc_id: str):
        """
        Invalide intelligemment le cache après mise à jour de document.
        """
        # Méthode 1 : Invalidation totale (simple mais brutal)
        # self.redis.flushdb()  # ❌ Trop agressif

        # Méthode 2 : Invalidation par pattern (mieux)
        # Trouver toutes les requêtes ayant retourné ce document
        affected_queries = self._find_queries_returning_doc(doc_id)
        for query in affected_queries:
            cache_key = self._generate_cache_key(query, k=10)
            self.redis.delete(cache_key)

        # Méthode 3 : Lazy invalidation (optimal)
        # Ajouter version/timestamp au cache
        doc_version = self._get_doc_version(doc_id)
        self.redis.hset(f"doc_versions", doc_id, doc_version)

    def get_with_version_check(self, cache_key: str, doc_ids: List[str]) ->
Optional[Dict]:
        """
        Vérifie que les versions des documents en cache sont à jour.
        """
        cached = self.redis.get(cache_key)
        if not cached:
            return None

        cached_data = json.loads(cached)
        cached_versions = cached_data.get('doc_versions', {})

        # Vérifier si les versions ont changé
        for doc_id in doc_ids:
            current_version = self.redis.hget("doc_versions", doc_id)
            if current_version != cached_versions.get(doc_id):
                # Version obsolète, invalider
                self.redis.delete(cache_key)
                return None

        return cached_data['results']
```

Approximate search pour la scalabilité

La recherche approximative (ANN - Approximate Nearest Neighbors) sacrifie légèrement la précision pour gagner 10-100x en vitesse et scalabilité.

Configuration HNSW pour Production

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

# Création de collection avec HNSW optimisé
client = QdrantClient(url="http://localhost:6333")

client.create_collection(
    collection_name="documents_prod",
    vectors_config=VectorParams(
        size=1024,  # Dimension des embeddings
        distance=Distance.COSINE
    ),
    hnsw_config=HnswConfigDiff(
        m=16,               # Nombre de connexions par noeud (défaut: 16)
        ef_construct=100,    # Qualité construction (défaut: 100)
        full_scan_threshold=10000  # Seuil full-scan (défaut: 10k)
    )
)

# Paramétrage recherche
results = client.search(
    collection_name="documents_prod",
    query_vector=query_embedding,
    limit=10,
    search_params={
        "hnsw_ef": 128,  # Qualité recherche (défaut: 128)
        "exact": False   # False = ANN rapide, True = exact lent
    }
)

# Trade-offs HNSW
"""
Paramètre   | Impact Latence | Impact Précision | Impact Mémoire
-----------------------------------------------------------------
m           | Faible         | Moyen            | Élevé
ef_construct| Aucun (index)  | Élevé            | Aucun
hnsw_ef     | Élevé          | Élevé            | Faible

Recommandations production :
- Corpus <1M   : m=16, ef_construct=100, hnsw_ef=64
- Corpus 1-10M : m=32, ef_construct=200, hnsw_ef=128 (défaut optimal)
- Corpus >10M  : m=48, ef_construct=400, hnsw_ef=256

Précision attendue : 95-99% vs exact search
Gain vitesse : 10-50x selon corpus
"""
```

Benchmarks ANN vs Exact

Corpus	Exact (ms)	HNSW (ms)	Speedup	Recall@10
100K docs	120	8	15x	99.2%
1M docs	850	25	34x	98.5%
10M docs	6,500	45	144x	97.8%
100M docs	65,000	80	812x	96.5%

Load balancing et réplication

Pour supporter 1000+ QPS, une architecture distribuée avec load balancing est nécessaire.

Architecture Multi-Noeud avec Qdrant

```
import random
from typing import List

class LoadBalancedSearchCluster:
    def __init__(self, qdrant_nodes: List[str]):
        """
        Args:
            qdrant_nodes: ["http://node1:6333", "http://node2:6333", "http://node3:6333"]
        """
        self.nodes = [QdrantClient(url=node) for node in qdrant_nodes]
        self.node_health = {i: True for i in range(len(self.nodes))}

    def search(self, query_vector: List[float], k: int = 10) -> List[Dict]:
        """
        Recherche avec load balancing et failover.
        """
        # Sélectionner un noeud sain aléatoirement (round-robin possible aussi)
        healthy_nodes = [i for i, healthy in self.node_health.items() if healthy]

        if not healthy_nodes:
            raise Exception("Aucun noeud sain disponible")

        node_idx = random.choice(healthy_nodes)

        try:
            results = self.nodes[node_idx].search(
                collection_name="documents",
                query_vector=query_vector,
                limit=k,
                timeout=5  # Timeout 5s
            )
            return results

        except Exception as e:
            # Marquer le noeud comme non sain
            self.node_health[node_idx] = False
            print(f"Noeud {node_idx} en erreur, failover...")

            # Retry sur un autre noeud
            remaining_nodes = [i for i in healthy_nodes if i != node_idx]
            if remaining_nodes:
                return self.search(query_vector, k)  # Recursion
            else:
                raise

    def health_check(self):
        """
        Vérification périodique de la santé des noeuds.
        À exécuter toutes les 10 secondes.
        """
        for i, client in enumerate(self.nodes):
            try:
                # Ping simple
                client.get_collections()
                self.node_health[i] = True
            except:
                self.node_health[i] = False

# Configuration Kubernetes pour auto-scaling
"""
apiVersion: apps/v1
```

```
kind: Deployment
metadata:
  name: qdrant-cluster
spec:
  replicas: 3  # 3 noeuds par défaut
  template:
    spec:
      containers:
      - name: qdrant
        image: qdrant/qdrant:latest
        resources:
          requests:
            memory: "8Gi"
            cpu: "2"
          limits:
            memory: "16Gi"
            cpu: "4"
---
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: qdrant-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: qdrant-cluster
  minReplicas: 3
  maxReplicas: 10
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 70
"""
```

Monitoring et alerting

Un monitoring proactif permet de détecter les problèmes avant qu'ils impactent les utilisateurs.

Voir section précédente sur "Monitoring en production" pour l'implémentation détaillée. Pour approfondir, consultez Qu'est-ce qu'un Embedding en.

Checklist Monitoring Production

- ✅ **Latence** : p50, p95, p99 (alertes si p95 >1s)
- ✅ **Throughput** : QPS actuel vs capacité max
- ✅ **Taux d'erreur** : Alerte si >1%
- ✅ **Cache hit rate** : Objectif >60%
- ✅ **Zero-results rate** : Objectif <10%
- ✅ **CTR** : Suivre quotidiennement (régression si -15%)
- ✅ **Santé des noeuds** : Disponibilité cluster >99.9%
- ✅ **Utilisation mémoire** : Alerte si >80%

Gestion des pics de charge

Les pics de charge (lancements produits, actualités virales) peuvent multiplier le trafic par 5-50x. Voici comment les absorber.

Stratégies Anti-Spike

1. Rate Limiting Adaptatif

```
from redis import Redis
import time

class AdaptiveRateLimiter:
    def __init__(self, redis_client: Redis):
        self.redis = redis_client

    def check_rate_limit(self, user_id: str, endpoint: str) -> bool:
        """
        Rate limiting avec buckets adaptatifs.
        """
        # Charge actuelle du système
        system_load = self._get_system_load()  # 0-100

        # Ajuster les limites selon la charge
        if system_load < 50:
            max_requests = 100  # Normal
        elif system_load < 80:
            max_requests = 50   # Surchargé
        else:
            max_requests = 10   # Critique

        # Token bucket algorithm
        key = f"rate_limit:{user_id}:{endpoint}"
        current = self.redis.incr(key)

        if current == 1:
            self.redis.expire(key, 60)  # Fenêtre 1 minute

        return current <= max_requests

    def _get_system_load(self) -> int:
        """Récupère la charge depuis métriques."""
        # Exemple : moyenne CPU/mémoire des noeuds
        return 42  # Simuler
```

2. Shedding de Charge (Load Shedding)

```
def search_with_degradation(query: str, user_tier: str) -> Dict:
    """
    Dégrade gracieusement le service selon la charge.
    """
    load = get_current_load_percentage()

    if load < 70:
        # Service complet : retrieval + reranking
        return full_search(query)

    elif load < 85:
        # Dégradation légère : skip reranking pour free users
        if user_tier == "premium":
            return full_search(query)
        else:
            return basic_search(query)  # Sans reranking

    else:
        # Dégradation forte : cache only
        cached = get_from_cache(query)
        if cached:
            return cached
        else:
            # Renvoyer résultats pré-calculés génériques
            return fallback_results(query)
```

3. Auto-Scaling Prédictif

- Monitorer les patterns de trafic (heures de pointe)
- Pré-scaler 15 minutes avant les pics prévisibles
- Garder 30% de capacité tampon pour absorber les pics imprévus
- Scale down progressif (pas brutal) pour éviter oscillations

Règles d'Or Production

- **Caching** : 60-80% hit rate réduit charge de 3-5x
- **ANN search** : HNSW avec 97%+ recall acceptable
- **Replication** : Minimum 3 noeuds pour HA
- **Rate limiting** : Adaptatif selon charge système
- **Degradation** : Préférer service dégradé que down complet
- **Monitoring** : Alertes proactives (p95 latence, error rate)

Sources et références : ArXiv IA · Hugging Face Papers

Articles connexes

- Embodied AI : Agents Physiques, Robotique et Sécurité en

Questions fréquentes

Le reranking est-il toujours nécessaire ?

Non, le reranking n'est pas toujours nécessaire, mais il apporte des gains significatifs dans la plupart des cas. Voici quand l'utiliser :

- ✅ **Recommandé** : Documentation technique, recherche légale/médicale, e-commerce, knowledge bases d'entreprise
- ⚠️ **Optionnel** : Contenu conversationnel simple, corpus très homogène, contraintes latence strictes (<100ms)
- ❌ **Inutile** : Corpus <1000 documents (overhead injustifié), recherche d'images/audio sans texte

Gains typiques : +10 à +20% sur NDCG@10, +15 à +30% sur le CTR. **Coût** : +150-350ms de latence supplémentaire. Le ROI est positif dès que la qualité de recherche impacte directement le business (conversions, productivité).

Quel impact sur la latence ?

Voici les latences typiques par composant sur un corpus de 10M documents :

- **Query embedding** : 20-50ms (modèle SentenceTransformer sur CPU)
- **Vector search (HNSW)** : 30-100ms (selon configuration ef)
- **Metadata filtering** : +5-20ms (selon sélectivité)
- **Reranking (100 docs)** : 200-500ms (cross-encoder sur GPU)
- **Hybrid search (BM25+vector)** : +50-100ms (exécution parallèle)

Latence totale end-to-end :

- Recherche simple : 50-150ms
- Recherche hybride : 100-250ms
- Recherche hybride + reranking : 250-650ms

Optimisations possibles : Caching (hit rate 60-80% → latence <5ms), GPU inference (reranking 3-5x plus rapide), reranking asynchrone (résultats progressifs).

Comment gérer les requêtes multilingues ?

Trois approches principales existent pour le multilingual :

1. Modèles Multilingual Natifs (Recommandé)

- **BGE-M3** : 100+ langues, excellent NDCG (0.84 multilingual MTEB)
- **multilingual-e5-large** : 94 langues, très performant
- **Cohere embed-multilingual-v3** : 100+ langues (API)

Avantage : Requête en français trouve docs en anglais/espagnol/etc. automatiquement.

2. Détection de Langue + Modèle Spécifique

```
from langdetect import detect

lang = detect(query)  # 'fr', 'en', 'es', etc.
if lang == 'fr':
    embedder = SentenceTransformer("camembert-base")
elif lang == 'en':
    embedder = SentenceTransformer("all-mpnet-base-v2")
# ...
```

Avantage : Meilleure précision par langue. **Inconvénient** : Pas de cross-lingual search.

3. Traduction Automatique
Traduire toutes les requêtes en anglais avant recherche. **Inconvénient** : Perte de nuances, latence +200ms, coût traduction.

Recommandation : Utilisez **BGE-M3** ou **multilingual-e5-large** pour 90% des cas. Réserve les modèles spécifiques aux langues avec très gros volume.

Peut-on combiner toutes ces techniques ?

Oui, et c'est même recommandé en production. Voici une stack complète optimale :

Pipeline Production Recommandé

1. **Query Processing** :
   - Correction orthographique (si nécessaire)
   - Expansion d'acronymes (domaine spécifique)
   - Query expansion avec LLM (pour requêtes courtes/vagues)
2. **Retrieval Hybride** :
   - Recherche dense (embeddings) : top-100
   - Recherche sparse (BM25) : top-100
   - Fusion RRF : top-100 unifiés
3. **Filtering** :
   - Pre-filtering (permissions, date range si sélectivité >10%)
   - Post-filtering (filtres très restrictifs)
4. **Reranking** :
   - Cross-encoder sur top-100 → top-10 final
   - Contextual boosting (user profile, session history)
5. **Post-Processing** :
   - Deduplication (si nécessaire)
   - Diversity reranking (MMR - Maximal Marginal Relevance)

Coût latence total : 300-700ms end-to-end. **Gains qualité** : NDCG@10 de 0.75 (baseline dense) → 0.93 (pipeline complet), soit +24%.

Optimisation : Exécuter retrieval dense et sparse en **parallèle** (asyncio) pour gagner 50-100ms. Utiliser **caching agressif** (hit rate 70%+) pour latence moyenne <150ms.

Comment mesurer l'amélioration concrète ?

Voici la méthodologie complète pour mesurer objectivement les améliorations :

1. Métriques Offline (Dataset d'Évaluation)

- **NDCG@10** : Métrique de référence (objectif : >0.85)
- **MRR** : Pour Q&A (objectif : >0.70)
- **Recall@100** : Pour retrieval (objectif : >0.90)

Processus : Constituer un dataset de 100-500 requêtes annotées manuellement (ou via LLM + validation). Réexécuter à chaque changement pour détecter régressions.

2. Métriques Online (Production)

- **CTR** (Click-Through Rate) : % utilisateurs cliquant sur un résultat
- **Time to click** : Temps avant premier clic (plus bas = mieux)
- **Zero-results rate** : % requêtes sans résultats (objectif : <10%)
- **Session success rate** : % sessions avec conversion/objectif atteint
- **User satisfaction** : Feedback explicite (thumbs up/down)

3. A/B Testing
Protocole :

1. Définir variantes (control vs traitement)
2. Assigner utilisateurs de manière stable (hash-based)
3. Collecter données pendant 2-4 semaines (minimum 10K recherches)
4. Analyse statistique (t-test, p-value <0.05 pour significativité)
5. Décision : rollout si gains >5% avec p<0.05

4. Exemple de Dashboard de Suivi

Métrique	Baseline (sem 1)	Après reranking (sem 2)	Δ	p-value
NDCG@10 (offline)	0.78	0.91	+16.7%	N/A
CTR	12.3%	15.8%	+28.5%	0.002
Time to click	8.5s	6.2s	-27.1%	0.018
Zero-results rate	14.2%	12.8%	-9.9%	0.125
Latence p95	120ms	420ms	+250%	N/A

Conclusion : Gains significatifs sur CTR et time-to-click (p<0.05), trade-off acceptable sur latence. **Décision** : Rollout à 100% avec optimisation latence en parallèle (caching, GPU).

Ressources open source associées :

- CUDAEmbeddings — Serveur d'embeddings GPU (Python)
- rag-langchain-fr — Dataset RAG & LangChain (HuggingFace)

Ayi NEDJIMI Consultants — Expert cybersécurité offensive & intelligence artificielle

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