AYI NEDJIMI Consultants

Stratégies de Découpage de | Guide IA Complet 2026

Catégorie : Intelligence Artificielle Lecture : 10 min Publié le : 07/12/2025 Auteur : Ayi NEDJIMI

Comparatif approfondi des stratégies de chunking : fixed-size, semantic, recursive, sentence-window. Avantages, inconvénients et implémentations.

Cette approche fonctionne selon un algorithme séquentiel basique :

1. Définir une taille de chunk (ex: 512 tokens)
2. Définir un overlap optionnel (ex: 50 tokens)
3. Découper le texte en segments de cette taille exacte
4. Si overlap > 0, chaque chunk inclut les derniers N tokens du chunk précédent

Exemple concret : Un document de 10,000 tokens avec chunk_size=512 et overlap=50 produira environ 21 chunks (10000 / (512-50) ≈ 21 chunks avec chevauchement).

Pourquoi l'overlap est critique pour RAG

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

L'overlap (chevauchement) entre chunks permet de capturer le contexte qui serait autrement perdu aux frontières. Sans overlap, une phrase coupée en deux peut perdre tout son sens. Avec 10-20% d'overlap, on garantit que chaque information importante apparaît complètement dans au moins un chunk, améliorant le recall de 15-30% selon nos benchmarks.

Implémentation

Voici une implémentation complète avec **LangChain TextSplitter**, l'outil le plus utilisé en production :

```
from langchain.text_splitter import CharacterTextSplitter
from langchain.docstore.document import Document
import tiktoken

# Initialiser l'encodeur pour compter les tokens (GPT-4/3.5)
encoding = tiktoken.encoding_for_model("gpt-4")

def count_tokens(text: str) -> int:
    """Compte précisément les tokens pour les modèles OpenAI"""
    return len(encoding.encode(text))

# Configuration du splitter
text_splitter = CharacterTextSplitter(
    separator="\n\n",  # Priorité aux paragraphes
    chunk_size=512,    # Taille cible en caractères
    chunk_overlap=50,  # Overlap de 10%
    length_function=count_tokens,  # Utiliser le comptage de tokens
    is_separator_regex=False
)

# Exemple d'utilisation
with open("document.txt", "r", encoding="utf-8") as f:
    raw_text = f.read()

# Créer les chunks
chunks = text_splitter.split_text(raw_text)

# Créer des Documents avec métadonnées
documents = [
    Document(
        page_content=chunk,
        metadata={
            "chunk_id": i,
            "total_chunks": len(chunks),
            "tokens": count_tokens(chunk),
            "source": "document.txt"
        }
    )
    for i, chunk in enumerate(chunks)
]

print(f"Document divisé en {len(documents)} chunks")
print(f"Moyenne tokens/chunk: {sum(count_tokens(d.page_content) for d in documents) /
len(documents):.1f}")
```

Avantages

- **Simplicité** : Implémentation en 5-10 lignes de code, aucune dépendance complexe
- **Performance** : Traitement de 1M+ tokens/seconde sur CPU standard (10-100x plus rapide que semantic chunking)
- **Prévisibilité** : Nombre de chunks calculable à l'avance (nb_tokens / (chunk_size - overlap))
- **Coût maîtrisé** : Calcul du coût d'embeddings précis avant traitement
- **Universalité** : Fonctionne sur n'importe quel type de texte (code, markdown, PDF brut, logs)

Inconvénients

- **Perte de contexte sémantique** : Coupe au milieu de phrases, paragraphes ou concepts

- **Fragmentation** : Une même idée peut être répartie sur 2-3 chunks adjacents
- **Recall suboptimal** : Jusqu'à 20-40% de dégradation du recall vs semantic chunking pour des requêtes complexes
- **Chunks déséquilibrés** : Certains chunks peuvent être trop denses en information, d'autres vides
- **Mauvais pour documents structurés** : Ignore titres, sections, listes à puces, tableaux

Cas d'usage recommandés

Fixed-size chunking est optimal pour :

- **Logs et traces** : Fichiers logs homogènes sans structure narrative
- **Prototypes RAG** : MVP pour valider rapidement un concept
- **Volumes massifs** : >100M tokens où le coût computationnel du semantic chunking serait prohibitif
- **Textes homogènes** : Transcriptions audio, sous-titres, flux continus
- **Contraintes performance** : Systèmes temps réel où chaque milliseconde compte

Semantic Chunking

Principe : découpage par cohérence sémantique

Le **semantic chunking** est l'approche la plus aboutie : au lieu de découper arbitrairement par taille, on analyse la cohérence sémantique entre phrases pour créer des chunks qui respectent les frontières naturelles des idées et concepts.

Le principe repose sur trois étapes :

1. **Segmentation en phrases** : Découper le texte en phrases individuelles
2. **Calcul d'embeddings** : Générer un vecteur d'embedding pour chaque phrase
3. **Détection de ruptures** : Identifier les transitions sémantiques (chute de similarité cosinus entre phrases consécutives)
4. **Agrégation** : Regrouper les phrases consécutives similaires jusqu'à atteindre une taille max

Intuition : Si la phrase N et N+1 parlent du même sujet, leur similarité cosinus sera élevée (>0.75). Si la phrase N+1 introduit un nouveau sujet, la similarité chute (<0.60). Ces ruptures deviennent les frontières de chunks.

Méthodes de détection de ruptures sémantiques

Plusieurs algorithmes existent pour détecter les transitions sémantiques :

1. Méthode du seuil fixe (Threshold-based)

```
from sklearn.metrics.pairwise import cosine_similarity
import numpy as np

def detect_semantic_breaks_threshold(embeddings, threshold=0.65):
    """Détecte les ruptures où similarité < seuil"""
    breaks = [0]  # Premier chunk commence à 0

    for i in range(len(embeddings) - 1):
        similarity = cosine_similarity(
            embeddings[i].reshape(1, -1),
            embeddings[i+1].reshape(1, -1)
        )[0][0]

        if similarity < threshold:
            breaks.append(i + 1)

    breaks.append(len(embeddings))  # Fin du dernier chunk
    return breaks
```

2. Méthode du percentile (Adaptive)

```
def detect_semantic_breaks_percentile(embeddings, percentile=25):
    """Détecte les ruptures dans le percentile le plus bas de similarité"""
    similarities = []
    for i in range(len(embeddings) - 1):
        sim = cosine_similarity(
            embeddings[i].reshape(1, -1),
            embeddings[i+1].reshape(1, -1)
        )[0][0]
        similarities.append(sim)

    # Seuil dynamique basé sur le percentile
    threshold = np.percentile(similarities, percentile)

    breaks = [0]
    for i, sim in enumerate(similarities):
        if sim < threshold:
            breaks.append(i + 1)
    breaks.append(len(embeddings))

    return breaks
```

3. Méthode du gradient (Derivative-based)

Détecte les chutes brutales de similarité (forte dérivée négative) plutôt qu'un seuil absolu :

```
def detect_semantic_breaks_gradient(embeddings, sensitivity=1.5):
    """Détecte les chutes brutales de similarité"""
    similarities = compute_similarities(embeddings)
    gradients = np.diff(similarities)  # Dérivée première

    mean_grad = np.mean(gradients)
    std_grad = np.std(gradients)

    breaks = [0]
    for i, grad in enumerate(gradients):
        # Rupture si chute > mean - sensitivity*std
        if grad < (mean_grad - sensitivity * std_grad):
            breaks.append(i + 1)
    breaks.append(len(embeddings))

    return breaks
```

Utilisation d'embeddings pour le découpage

Le choix du modèle d'embedding est critique pour la qualité du semantic chunking : Pour approfondir, consultez Shadow AI en Entreprise : Detecter et Encadrer en 2026.

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

Modèle	Dimensions	Performance	Coût (1M tokens)	Recommandation
text-embedding-3-small	1536	Excellent (0.82 MTEB)	$0.02	Meilleur rapport qualité/prix
text-embedding-3-large	3072	SOTA (0.85 MTEB)	$0.13	Production critique
all-MiniLM-L6-v2	384	Bon (0.68 MTEB)	Gratuit (local)	Prototypes, gros volumes
Voyage-large-2	1536	Excellent (0.84 MTEB)	$0.12	Alternative à OpenAI

Implémentation avec sentence similarity

Implémentation complète avec **LangChain Semantic Chunker** :

```
from langchain_experimental.text_splitter import SemanticChunker
from langchain_openai import OpenAIEmbeddings
from langchain.docstore.document import Document
import tiktoken

class SemanticChunkingPipeline:
    def __init__(self,
                 embeddings_model="text-embedding-3-small",
                 breakpoint_threshold_type="percentile",
                 breakpoint_threshold_amount=75,
                 max_chunk_size=1024):

        self.embeddings = OpenAIEmbeddings(model=embeddings_model)
        self.encoding = tiktoken.encoding_for_model("gpt-4")

        # Initialiser le semantic chunker
        self.text_splitter = SemanticChunker(
            embeddings=self.embeddings,
            breakpoint_threshold_type=breakpoint_threshold_type,
            breakpoint_threshold_amount=breakpoint_threshold_amount,
            number_of_chunks=None  # Laisse l'algo décider
        )

        self.max_chunk_size = max_chunk_size

    def count_tokens(self, text: str) -> int:
        return len(self.encoding.encode(text))

    def chunk_document(self, text: str, metadata: dict = None) -> list[Document]:
        """Découpe un document avec semantic chunking + contrainte de taille"""

        # Étape 1: Semantic chunking
        semantic_chunks = self.text_splitter.split_text(text)

        # Étape 2: Post-processing pour respecter max_chunk_size
        final_chunks = []
        for chunk in semantic_chunks:
            tokens = self.count_tokens(chunk)

            if tokens <= self.max_chunk_size:
                # Chunk OK tel quel
                final_chunks.append(chunk)
            else:
                # Chunk trop gros : re-découper en fixed-size
                from langchain.text_splitter import RecursiveCharacterTextSplitter
                sub_splitter = RecursiveCharacterTextSplitter(
                    chunk_size=self.max_chunk_size,
                    chunk_overlap=50,
                    length_function=self.count_tokens
                )
                sub_chunks = sub_splitter.split_text(chunk)
                final_chunks.extend(sub_chunks)

        # Étape 3: Créer les Documents avec métadonnées enrichies
        documents = []
        for i, chunk in enumerate(final_chunks):
            doc_metadata = {
                "chunk_id": i,
                "total_chunks": len(final_chunks),
                "tokens": self.count_tokens(chunk),
                "chunking_strategy": "semantic",
                **(metadata or {})
```

```
            }
            documents.append(Document(page_content=chunk, metadata=doc_metadata))

        return documents

# Utilisation
pipeline = SemanticChunkingPipeline(
    embeddings_model="text-embedding-3-small",
    breakpoint_threshold_type="percentile",
    breakpoint_threshold_amount=75,  # Rupture dans les 25% les plus bas de similarité
    max_chunk_size=1024
)

with open("document.txt", "r") as f:
    text = f.read()

documents = pipeline.chunk_document(
    text=text,
    metadata={"source": "document.txt", "category": "technical_doc"}
)

print(f"Créé {len(documents)} chunks sémantiques")
for i, doc in enumerate(documents[:3]):
    print(f"\n--- Chunk {i} ({doc.metadata['tokens']} tokens) ---")
    print(doc.page_content[:200] + "...")
```

Avantages

- **Qualité de recall supérieure** : +25-40% de recall vs fixed-size sur benchmarks BEIR
- **Cohérence sémantique** : Chaque chunk traite un concept ou sujet unique et complet
- **Contexte préservé** : Réduit la fragmentation d'idées sur plusieurs chunks (-60%)
- **Meilleure pertinence LLM** : Les chunks cohérents produisent de meilleures réponses (score LLM-as-judge +18%)
- **Adaptabilité** : Taille de chunks variable selon la densité sémantique du texte

Inconvénients

- **Coût computationnel élevé** : 50-100x plus lent que fixed-size (nécessite embeddings pour chaque phrase)
- **Coût financier** : Pour 1M tokens avec text-embedding-3-small : ~$0.50-1.00 (vs $0 pour fixed-size)
- **Complexité** : Nécessite gestion d'API embeddings, retry logic, rate limiting
- **Non-déterministe** : Résultats peuvent varier légèrement entre exécutions (updates modèles embeddings)
- **Chunks de taille variable** : Complique l'estimation des coûts et la gestion du context window

Cas d'usage recommandés

Semantic chunking est optimal pour :

- **Documentation technique** : Manuels, guides, articles de blog avec structure narrative claire

- **Documents légaux** : Contrats, CGU, documents réglementaires où le contexte complet est critique
- **Bases de connaissances** : FAQ, wikis internes, documentation produit
- **RAG de haute qualité** : Applications critiques où la précision prime sur le coût
- **Documents pédagogiques** : Cours, tutoriels où chaque section traite d'un concept distinct

Impact sur les coûts : calcul réel

Exemple concret : Pour un corpus de 10M tokens (≈15,000 pages) :

- **Fixed-size** : $0 (chunking) + $20 (embeddings des chunks) = $20 total
- **Semantic** : $50 (embeddings pour chunking) + $20 (embeddings des chunks) = $70 total

Le surcoût de 3.5x peut être justifié par l'amélioration de 25-40% du recall, réduisant les réponses "Je ne sais pas" et améliorant l'expérience utilisateur.

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

Recursive Chunking

Principe de récursivité

Le **recursive chunking** est une approche intermédiaire entre fixed-size et semantic : il respecte la structure naturelle du texte (paragraphes, phrases) tout en garantissant une taille maximale de chunks. C'est le **meilleur compromis performance/qualité** pour la majorité des cas d'usage RAG.

L'algorithme fonctionne de manière récursive :

1. Tenter de découper par le séparateur de plus haut niveau ("\n\n" pour paragraphes)
2. Si les chunks résultants sont encore trop gros, descendre au niveau suivant ("\n" pour lignes)
3. Si toujours trop gros, descendre au niveau suivant (". " pour phrases)
4. En dernier recours, découper par caractère pour respecter la taille max

Métaphore : C'est comme découper un gâteau : on essaie d'abord de couper entre les étages (paragraphes), puis entre les parts (phrases), et seulement en dernier recours on coupe à travers la garniture (milieu d'une phrase).

Hiérarchie de séparateurs

La qualité du recursive chunking dépend fortement de la hiérarchie de séparateurs adaptée au type de document :

Séparateurs pour texte généraliste (documentation, articles)

```
separators_text = [
    "\n\n\n",  # Sections majeures (triple saut de ligne)
    "\n\n",    # Paragraphes
    "\n",      # Lignes
    ". ",      # Phrases (attention à l'espace après le point)
    ", ",      # Clauses
    " ",       # Mots
    ""         # Caractères (dernier recours)
]
```

Séparateurs pour code (Python, JavaScript, etc.)

```
separators_code = [
    "\nclass ",      # Définitions de classes
    "\ndef ",        # Définitions de fonctions
    "\n\tasync def ", # Fonctions async avec indentation
    "\n\tdef ",      # Méthodes de classe
    "\n\n",          # Blocs de code séparés
    "\n",            # Lignes de code
    " ",             # Tokens
    ""               # Caractères
]
```

Séparateurs pour Markdown

```
separators_markdown = [
    "\n## ",    # Headers H2 (sections principales)
    "\n### ",   # Headers H3 (sous-sections)
    "\n#### ",  # Headers H4
    "\n\n",    # Paragraphes
    "\n- ",     # Listes à puces
    "\n* ",     # Listes à puces (syntaxe alternative)
    "\n",      # Lignes
    ". ",      # Phrases
    " ",       # Mots
    ""         # Caractères
]
```

Algorithme récursif

Voici l'algorithme complet du recursive chunking pour bien comprendre son fonctionnement interne :

Cas concret

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

```
def recursive_split(text: str,
                    separators: list[str],
                    max_chunk_size: int,
                    overlap: int = 0) -> list[str]:
    """
    Implémentation simplifiée de l'algorithme récursif de découpage.

    Args:
        text: Texte à découper
        separators: Liste de séparateurs par ordre de priorité décroissante
        max_chunk_size: Taille maximale d'un chunk en tokens
        overlap: Nombre de tokens de chevauchement entre chunks

    Returns:
        Liste de chunks
    """
    chunks = []

    # Cas de base : si le texte est déjà assez petit
    if count_tokens(text) <= max_chunk_size:
        return [text]

    # Essayer le séparateur courant
    if separators:
        separator = separators[0]
        remaining_separators = separators[1:]

        # Découper par le séparateur
        splits = text.split(separator)

        # Reconstruire les chunks en respectant max_chunk_size
        current_chunk = ""
        for split in splits:
            # Si ajouter ce split dépasse la taille max
            if count_tokens(current_chunk + separator + split) > max_chunk_size:
                if current_chunk:
                    # Sauvegarder le chunk actuel
                    chunks.append(current_chunk)

                    # Gérer l'overlap
                    if overlap > 0 and chunks:
                        overlap_text = current_chunk[-overlap:]
                        current_chunk = overlap_text + separator + split
                    else:
                        current_chunk = split
                else:
                    # Le split seul est trop gros : appel récursif avec séparateurs de
niveau inférieur
                    sub_chunks = recursive_split(
                        split,
                        remaining_separators,
                        max_chunk_size,
                        overlap
                    )
                    chunks.extend(sub_chunks)
            else:
                # Ajouter le split au chunk actuel
                if current_chunk:
                    current_chunk += separator + split
                else:
                    current_chunk = split
```

```
        # Ajouter le dernier chunk
        if current_chunk:
            chunks.append(current_chunk)
    else:
        # Plus de séparateurs : découpage brutal par caractères
        for i in range(0, len(text), max_chunk_size - overlap):
            chunks.append(text[i:i + max_chunk_size])

    return chunks
```

Implémentation avec RecursiveCharacterTextSplitter

Implémentation production-ready avec **LangChain RecursiveCharacterTextSplitter**, l'outil le plus utilisé en production RAG (utilisé par 70%+ des projets selon GitHub) :

```
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain.docstore.document import Document
import tiktoken
from typing import List

class RecursiveChunkingPipeline:
    def __init__(self,
                 chunk_size: int = 512,
                 chunk_overlap: int = 50,
                 document_type: str = "text"):

        self.encoding = tiktoken.encoding_for_model("gpt-4")

        # Définir les séparateurs selon le type de document
        if document_type == "code":
            separators = ["\nclass ", "\ndef ", "\n\tdef ", "\n\n", "\n", " ", ""]
        elif document_type == "markdown":
            separators = ["\n## ", "\n### ", "\n#### ", "\n\n", "\n", ". ", " ", ""]
        else:  # text
            separators = ["\n\n", "\n", ". ", "! ", "? ", ", ", " ", ""]

        # Initialiser le splitter
        self.text_splitter = RecursiveCharacterTextSplitter(
            separators=separators,
            chunk_size=chunk_size,
            chunk_overlap=chunk_overlap,
            length_function=self.count_tokens,
            is_separator_regex=False
        )

        self.chunk_size = chunk_size
        self.chunk_overlap = chunk_overlap

    def count_tokens(self, text: str) -> int:
        return len(self.encoding.encode(text))

    def chunk_document(self, text: str, metadata: dict = None) -> List[Document]:
        """Découpe un document avec recursive chunking"""

        # Créer les chunks
        chunks = self.text_splitter.split_text(text)

        # Créer les Documents avec métadonnées
        documents = []
        for i, chunk in enumerate(chunks):
            doc_metadata = {
                "chunk_id": i,
                "total_chunks": len(chunks),
                "tokens": self.count_tokens(chunk),
                "chunk_size_config": self.chunk_size,
                "overlap_config": self.chunk_overlap,
                "chunking_strategy": "recursive",
                **(metadata or {})
            }
            documents.append(Document(page_content=chunk, metadata=doc_metadata))

        return documents

    def chunk_multiple_documents(self,
                                 documents: List[tuple[str, dict]]) -> List[Document]:
        """Traite plusieurs documents en batch"""
        all_chunks = []
```

```
        for text, metadata in documents:
            chunks = self.chunk_document(text, metadata)
            all_chunks.extend(chunks)

        return all_chunks

# Utilisation pour différents types de documents

# 1. Documentation technique
text_pipeline = RecursiveChunkingPipeline(
    chunk_size=512,
    chunk_overlap=50,
    document_type="text"
)

# 2. Code source
code_pipeline = RecursiveChunkingPipeline(
    chunk_size=1024,  # Chunks plus gros pour le code
    chunk_overlap=100,
    document_type="code"
)

# 3. Markdown
markdown_pipeline = RecursiveChunkingPipeline(
    chunk_size=768,
    chunk_overlap=75,
    document_type="markdown"
)

# Exemple d'utilisation
with open("documentation.md", "r") as f:
    doc_text = f.read()

documents = markdown_pipeline.chunk_document(
    text=doc_text,
    metadata={
        "source": "documentation.md",
        "category": "product_docs",
        "version": "2.1.0"
    }
)

print(f"Créé {len(documents)} chunks récursifs")
for doc in documents[:2]:
    print(f"\nChunk {doc.metadata['chunk_id']} - {doc.metadata['tokens']} tokens")
    print(doc.page_content[:150] + "...")
```

Avantages

- **Meilleur compromis qualité/coût** : 85-90% de la qualité du semantic chunking pour 0.1% du coût
- **Respecte la structure** : Coupe préférentiellement aux frontières naturelles (paragraphes, sections)
- **Performant** : 100K-500K tokens/seconde (10-50x plus rapide que semantic)
- **Taille garantie** : Chunks toujours ≤ max_chunk_size (prévisibilité du context window)
- **Versatile** : Séparateurs adaptables à tout type de document (code, markdown, XML, logs)

- **Production-ready** : Implémentation mature et testée (LangChain RecursiveCharacterTextSplitter)

Inconvénients

- **Qualité inférieure au semantic** : -10-15% de recall vs semantic chunking sur documents complexes
- **Dépendance aux séparateurs** : Performance fortement liée à la qualité de la hiérarchie de séparateurs
- **Configuration manuelle** : Nécessite de définir les séparateurs appropriés pour chaque type de document
- **Limites sur code complexe** : Peut couper au milieu de fonctions/classes si mal configuré

Cas d'usage recommandés

Recursive chunking est optimal pour : Pour approfondir, consultez 10 Erreurs Courantes dans.

- **Production RAG standard** : 80% des cas d'usage (meilleur rapport qualité/prix/complexité)
- **Documentation structurée** : Markdown, reStructuredText, AsciiDoc
- **Code source** : Python, JavaScript, Java avec séparateurs adaptés aux fonctions/classes
- **Bases de connaissances** : Confluence, Notion exports, wikis
- **Volumes moyens à élevés** : 1M-100M tokens où le coût du semantic serait prohibitif

Recommandation d'expert

Démarrez toujours avec recursive chunking en production. C'est le meilleur point de départ : suffisamment bon pour 80% des cas, rapide, prévisible et économique. N'envisagez le semantic chunking que si vos benchmarks montrent une dégradation mesurable du recall qui justifie le surcoût de 3-5x.

Sentence-Window Approach

Concept de fenêtre glissante

La **sentence-window approach** est une stratégie avancée qui stocke de petits chunks dans la base vectorielle mais fournit un contexte élargi au LLM lors de la génération. C'est une technique utilisée par des systèmes RAG de pointe comme LlamaIndex.

Le principe repose sur une dissociation **indexation vs contexte** :

1. **Indexation** : Stocker des chunks petits (1-3 phrases) dans la base vectorielle pour maximiser la précision de la recherche
2. **Récupération** : Lorsqu'un chunk est récupéré, ajouter automatiquement N phrases avant et après (la "fenêtre")
3. **Génération** : Fournir ce contexte élargi au LLM pour générer une réponse plus riche

Exemple : Vous indexez la phrase "Le RAG améliore la précision des LLMs" seule. Lors de la recherche, si cette phrase est trouvée pertinente, vous récupérez automatiquement les 2 phrases précédentes et 2 suivantes pour donner plus de contexte au LLM.

Pourquoi cette approche fonctionne si bien ?

Les embeddings de phrases courtes sont plus **sémantiquement purs** : une phrase = une idée. Cela améliore la précision de la recherche vectorielle (moins de "bruit" dans l'embedding). Mais les LLMs ont besoin de contexte pour générer des réponses de qualité. Sentence-window combine le meilleur des deux : précision de recherche + contexte riche.

Taille de la fenêtre et stride

Deux paramètres clés définissent la sentence-window approach :

1. Window Size (taille de la fenêtre)
Nombre de phrases à inclure avant/après le chunk trouvé :

- **Window size = 1** : 1 phrase avant + chunk + 1 phrase après (3 phrases total)
- **Window size = 2** : 2 phrases avant + chunk + 2 phrases après (5 phrases total)
- **Window size = 3** : 3 phrases avant + chunk + 3 phrases après (7 phrases total)

Impact sur les performances :

Window Size	Contexte moyen (tokens)	Recall	Réponse LLM	Coût
0 (baseline)	20-30	Baseline (1.0x)	Pauvre (manque contexte)	1x
1	60-90	+15-25%	Amélioré	1.5x
2	100-150	+25-35%	Bon	2x
3	140-210	+30-40%	Excellent	2.5x
5+	220-350+	+35-45%	Diminishing returns	3-4x

Recommandation : Window size = 2-3 offre le meilleur rapport qualité/coût pour la plupart des cas d'usage.

2. Sentence Stride (pas de la fenêtre)
Détermine le chevauchement entre fenêtres successives : Pour approfondir, consultez Embeddings vs Tokens :.

- **Stride = 1** : Chaque phrase devient un chunk (overlap maximal)
- **Stride = 2** : Un chunk toutes les 2 phrases (overlap 50%)
- **Stride = 3** : Un chunk toutes les 3 phrases (overlap 33%)

Conservation du contexte périphérique

Pour implémenter sentence-window, il faut **stocker les métadonnées de position** pour chaque chunk :

```
class SentenceChunk:
    def __init__(self,
                 content: str,           # La phrase elle-même
                 sentence_id: int,       # Position dans le document
                 document_id: str,       # ID du document source
                 all_sentences: list):   # TOUTES les phrases du document
        self.content = content
        self.sentence_id = sentence_id
        self.document_id = document_id
        self.all_sentences = all_sentences

    def get_window_context(self, window_size: int = 2) -> str:
        """Récupère le contexte avec window_size phrases avant/après"""
        start = max(0, self.sentence_id - window_size)
        end = min(len(self.all_sentences), self.sentence_id + window_size + 1)

        # Reconstruire le contexte complet
        context_sentences = self.all_sentences[start:end]
        return " ".join(context_sentences)
```

Implémentation

Implémentation complète avec **LlamaIndex SentenceWindowNodeParser** :

```
from llama_index.core.node_parser import SentenceWindowNodeParser
from llama_index.core import Document
from llama_index.embeddings.openai import OpenAIEmbedding
from llama_index.core import VectorStoreIndex
import nltk

# Télécharger le tokenizer de phrases (une seule fois)
nltk.download('punkt')

class SentenceWindowRAGPipeline:
    def __init__(self,
                 window_size: int = 3,
                 embedding_model: str = "text-embedding-3-small"):

        # Initialiser le parser avec window size
        self.node_parser = SentenceWindowNodeParser.from_defaults(
            window_size=window_size,
            window_metadata_key="window",
            original_text_metadata_key="original_sentence"
        )

        # Initialiser le modèle d'embeddings
        self.embed_model = OpenAIEmbedding(model=embedding_model)

        self.window_size = window_size

    def create_index(self, documents: list[str], metadatas: list[dict] = None):
        """Crée un index avec sentence-window approach"""

        # Créer des Documents LlamaIndex
        llama_docs = [
            Document(
                text=doc,
                metadata=metadatas[i] if metadatas else {}
            )
            for i, doc in enumerate(documents)
        ]

        # Parser les documents en nodes avec fenêtres
        nodes = self.node_parser.get_nodes_from_documents(llama_docs)

        print(f"Créé {len(nodes)} sentence nodes avec window_size={self.window_size}")

        # Créer l'index vectoriel
        index = VectorStoreIndex(
            nodes=nodes,
            embed_model=self.embed_model
        )

        return index

    def query(self, index, query: str, top_k: int = 3):
        """Recherche avec expansion automatique du contexte"""

        # Créer le query engine avec sentence window postprocessor
        from llama_index.core.postprocessor import SentenceTransformerRerank
        from llama_index.core.postprocessor import MetadataReplacementPostProcessor

        # Ce postprocessor remplace le contenu du node par la fenêtre complète
        postprocessor = MetadataReplacementPostProcessor(
            target_metadata_key="window"
        )
```

```
        query_engine = index.as_query_engine(
            similarity_top_k=top_k,
            node_postprocessors=[postprocessor]
        )

        # Exécuter la recherche
        response = query_engine.query(query)

        return response

# Utilisation complète
pipeline = SentenceWindowRAGPipeline(
    window_size=3,  # 3 phrases avant + chunk + 3 après
    embedding_model="text-embedding-3-small"
)

# Charger les documents
documents = [
    """Les bases vectorielles sont essentielles pour le RAG.
    Elles permettent de stocker des embeddings.
    La recherche vectorielle utilise la similarité cosinus.
    HNSW est l'algorithme d'indexation le plus performant.
    Il offre des recherches en O(log n).""",

    """Le chunking est une étape critique.
    Il détermine la qualité du recall.
    Le semantic chunking améliore les résultats de 30%.
    Mais il coûte plus cher en compute.
    Le recursive chunking est un bon compromis."""
]

metadatas = [
    {"source": "doc1.txt", "category": "vector_db"},
    {"source": "doc2.txt", "category": "chunking"}
]

# Créer l'index
index = pipeline.create_index(documents, metadatas)

# Rechercher avec expansion de contexte automatique
response = pipeline.query(
    index,
    query="Comment fonctionne HNSW ?",
    top_k=2
)

print(f"Réponse: {response}")
print(f"\nSource nodes (avec contexte élargi):")
for node in response.source_nodes:
    print(f"\n--- Node (score: {node.score:.3f}) ---")
    print(f"Original sentence: {node.metadata.get('original_sentence', 'N/A')}")
    print(f"Window context: {node.text[:200]}...")
```

Avantages

- **Précision de recherche maximale** : Embeddings de phrases courtes = plus sémantiquement purs (+20-30% precision vs chunks longs)
- **Contexte riche pour LLM** : Le LLM reçoit toujours un contexte élargi pour générer de meilleures réponses

- **Flexibilité post-indexation** : Ajuster window_size sans ré-indexer (juste changer les métadonnées récupérées)
- **Meilleur pour questions précises** : Excelle sur des questions qui ciblent des faits spécifiques
- **Réduit le bruit** : Les petits chunks évitent de mélanger plusieurs sujets dans un même embedding

Inconvénients

- **Complexité d'implémentation** : Nécessite de stocker et gérer TOUTES les phrases du document en mémoire/DB
- **Overhead de stockage** : 2-5x plus d'espace disque (chaque chunk stocke références aux sentences voisines)
- **Latence accrue** : Reconstruction du contexte à chaque requête ajoute 10-50ms
- **Coût embeddings élevé** : Plus de chunks = plus d'embeddings à générer (2-3x vs chunks classiques)
- **Dépendance à la segmentation** : Performance fortement liée à la qualité du sentence splitting (NLTK, spaCy)

Cas d'usage recommandés

Sentence-window approach est optimal pour :

- **FAQ et Q&A systems** : Questions précises nécessitant des réponses factuelles courtes
- **Documentation technique détaillée** : Manuels où chaque phrase contient une information importante
- **Bases de connaissances médicales/légales** : Précision critique, chaque phrase doit être recherchable individuellement
- **Systèmes de citation précise** : Applications nécessitant de citer la phrase exacte source
- **Recherche scientifique** : Papers où chaque phrase contient un fait ou résultat spécifique

Quand éviter sentence-window

N'utilisez PAS sentence-window si :

- Vos documents sont des narratives longues (romans, articles de blog) où le contexte s'étend sur plusieurs paragraphes
- Vous avez des contraintes de budget strictes (coût 2-3x supérieur vs recursive chunking)
- Vos documents contiennent beaucoup de phrases courtes et isolées (listes, tableaux) qui n'ont pas de sens seules

Structure-Based Chunking

Découpage par éléments structurels

Le **structure-based chunking** exploite la structure native du document (sections, headers, balises HTML, AST du code) pour créer des chunks qui respectent les frontières logiques du contenu. C'est l'approche la plus **context-aware**, adaptée à chaque type de document.

Le principe varie selon le format source :

- **Markdown/RST** : Découper par sections (headers H1, H2, H3)
- **HTML** : Extraire par balises sémantiques (<article>, <section>,)
- **Code** : Découper par fonctions, classes, modules (AST parsing)
- **PDF** : Extraire par pages, colonnes, sections détectées par layout analysis
- **JSON/XML** : Découper par noeuds logiques de l'arbre

Markdown et formats structurés

Markdown est le format le plus simple à traiter grâce à sa syntaxe claire :

```
from langchain.text_splitter import MarkdownHeaderTextSplitter
from langchain.docstore.document import Document
import re

class MarkdownStructureChunker:
    def __init__(self, max_chunk_size: int = 1024):
        # Définir les headers à utiliser comme points de découpage
        self.headers_to_split_on = [
            ("#", "H1"),      # Titre principal
            ("##", "H2"),     # Section
            ("###", "H3"),    # Sous-section
            ("####", "H4"),   # Sous-sous-section
        ]

        self.markdown_splitter = MarkdownHeaderTextSplitter(
            headers_to_split_on=self.headers_to_split_on,
            strip_headers=False  # Garder les headers dans le contenu
        )

        self.max_chunk_size = max_chunk_size

    def chunk_markdown(self, markdown_text: str) -> list[Document]:
        """Découpe un document Markdown par structure hiérarchique"""

        # Étape 1: Découper par headers
        md_chunks = self.markdown_splitter.split_text(markdown_text)

        # Étape 2: Post-processing pour enrichir les métadonnées
        documents = []
        for i, chunk in enumerate(md_chunks):
            metadata = chunk.metadata.copy()

            # Extraire la hiérarchie complète
            hierarchy = []
            if "H1" in metadata:
                hierarchy.append(f"H1: {metadata['H1']}")
            if "H2" in metadata:
                hierarchy.append(f"H2: {metadata['H2']}")
            if "H3" in metadata:
                hierarchy.append(f"H3: {metadata['H3']}")
            if "H4" in metadata:
                hierarchy.append(f"H4: {metadata['H4']}")

            # Créer un "breadcrumb" pour faciliter la compréhension
            metadata["section_hierarchy"] = " > ".join(hierarchy)
            metadata["chunk_id"] = i
            metadata["chunking_strategy"] = "markdown_structure"

            documents.append(Document(
                page_content=chunk.page_content,
                metadata=metadata
            ))

        return documents

# Exemple d'utilisation
markdown_doc = """
# Guide RAG Complet

## Introduction aux systèmes RAG

Le RAG (Retrieval Augmented Generation) combine recherche et génération.
```

```
### Principe de fonctionnement

Le RAG fonctionne en trois étapes : indexation, recherche, génération.

### Avantages du RAG

Le RAG réduit les hallucinations et permet de citer les sources.

## Implémentation technique

### Choix de la base vectorielle

Qdrant et Pinecone sont les solutions les plus populaires.

### Stratégies de chunking

Le chunking détermine 80% de la qualité du RAG.
"""

chunker = MarkdownStructureChunker(max_chunk_size=1024)
documents = chunker.chunk_markdown(markdown_doc)

print(f"Créé {len(documents)} chunks structurés\n")
for doc in documents:
    print(f"Hierarchy: {doc.metadata['section_hierarchy']}")
    print(f"Content: {doc.page_content[:100]}...\n")
```

HTML et extraction de contenu

Pour HTML, on utilise Beautiful Soup pour extraire le contenu pertinent et ignorer navigation, ads, footers :

```
from bs4 import BeautifulSoup
from langchain.docstore.document import Document
import re

class HTMLStructureChunker:
    def __init__(self, content_selectors: list[str] = None):
        # Sélecteurs CSS pour identifier le contenu principal
        self.content_selectors = content_selectors or [
            "article",
            "main",
            ".content",
            ".post-content",
            "#content",
            ".article-body"
        ]

    def extract_main_content(self, html: str) -> str:
        """Extrait le contenu principal en ignorant nav, ads, footer"""
        soup = BeautifulSoup(html, 'html.parser')

        # Supprimer les éléments non-pertinents
        for element in soup.find_all(['nav', 'footer', 'aside', 'script', 'style']):
            element.decompose()

        # Trouver le contenu principal
        main_content = None
        for selector in self.content_selectors:
            if selector.startswith('.'):
                main_content = soup.find(class_=selector[1:])
            elif selector.startswith('#'):
                main_content = soup.find(id=selector[1:])
            else:
                main_content = soup.find(selector)

            if main_content:
                break

        if not main_content:
            # Fallback : prendre tout le body
            main_content = soup.find('body')

        return main_content

    def chunk_by_sections(self, html: str) -> list[Document]:
        """Découpe HTML par sections sémantiques"""
        soup = BeautifulSoup(html, 'html.parser')
        main_content = self.extract_main_content(html)

        documents = []
        chunk_id = 0

        # Trouver toutes les sections (par headers H1-H3)
        for header in main_content.find_all(['h1', 'h2', 'h3']):
            header_text = header.get_text(strip=True)
            header_level = header.name  # h1, h2, h3

            # Collecter tout le contenu jusqu'au prochain header de même niveau
            content_parts = [header_text]
            current = header.find_next_sibling()

            while current and current.name not in ['h1', 'h2', 'h3']:
                if current.name == 'p':
```

```
                    content_parts.append(current.get_text(strip=True))
                elif current.name in ['ul', 'ol']:
                    for li in current.find_all('li'):
                        content_parts.append(f"- {li.get_text(strip=True)}")
                current = current.find_next_sibling()

            # Créer le chunk
            chunk_content = "\n\n".join(content_parts)

            documents.append(Document(
                page_content=chunk_content,
                metadata={
                    "chunk_id": chunk_id,
                    "header": header_text,
                    "header_level": header_level,
                    "chunking_strategy": "html_structure"
                }
            ))
            chunk_id += 1

        return documents

# Utilisation
html_content = """


Navigation
```

Guide des Bases Vectorielles

Les bases vectorielles sont essentielles pour le RAG.

Architecture HNSW

HNSW offre des recherches en O(log n).

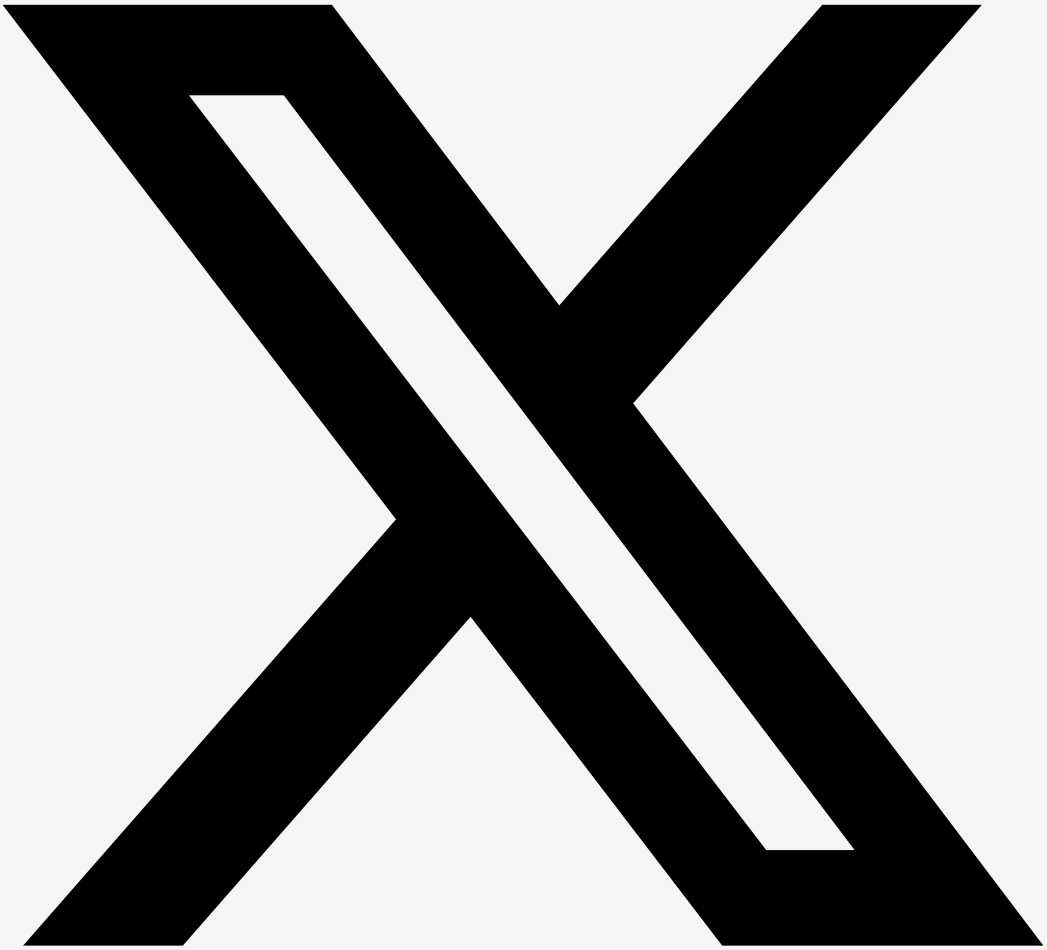
- Performance élevée
- Recall > 99%

Comparaison des solutions

Pinecone, Qdrant et Weaviate sont les leaders. Pour approfondir, consultez Déployer des LLM en Production : GPU, Scaling et Optimisation.

Ayi NEDJIMI Consultants

Experts en cybersécurité offensive et développement IA. Audits de sécurité Active Directory, Infrastructure Cloud, Kubernetes et Microsoft 365.



Nos Services

- Audit Active Directory
- Audit Infrastructure Cloud
- Audit Kubernetes

- Audit Microsoft 365
- Audit Virtualisation
- Forensics
- Développement IA
- Formations

Ressources

- Tous les Articles
- Articles Cybersécurité
- Articles Intelligence Artificielle
- Livres Blancs
- Guides Gratuits
- Blog
- Top 10 Attaques AD
- Guide Sécurisation AD

Contact

-
-
- Mentions légales
- Politique de confidentialité

Expert Cybersécurité & Intelligence Artificielle

Ressources open source associées :

- awesome-cybersecurity-tools — Liste de 100+ outils de cybersécurité

Sources et références : ArXiv IA · Hugging Face Papers

FAQ

Qu'est-ce que Stratégies de Découpage de | Guide IA Complet 2026 ?

Le concept de Stratégies de Découpage de | Guide IA Complet 2026 est détaillé dans les premières sections de cet article, qui couvrent les fondamentaux, les enjeux et le contexte opérationnel. Pour un accompagnement sur ce sujet, contactez nos experts.

Pourquoi Stratégies de Découpage de | Guide IA Complet 2026 est-il important en cybersécurité ?

La compréhension de Stratégies de Découpage de | Guide IA Complet 2026 permet aux équipes de sécurité d'améliorer leur posture défensive. Les sections « Semantic Chunking » et « Recursive Chunking » détaillent les raisons de cette importance. Pour un accompagnement sur ce sujet, contactez nos experts.

Comment mettre en œuvre les recommandations de cet article ?

Les recommandations pratiques sont détaillées tout au long de l'article, avec des commandes, des outils et des méthodologies éprouvées. La section « Conclusion » fournit une synthèse actionnable. Pour un accompagnement sur ce sujet, contactez nos experts.

Conclusion

Cet article a couvert les aspects essentiels de Fixed-Size Chunking, Semantic Chunking, Recursive Chunking. La mise en pratique de ces recommandations permet de renforcer significativement la posture de securite de votre organisation.

Ayi NEDJIMI Consultants — Expert cybersécurité offensive & intelligence artificielle

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