

Analysis of Chunking Strategies for LLM Applications and proposal of new strategy

Msc Aderiye Tayo

Master's Thesis
2025



Tomas Bata University in Zlín
Faculty of Applied Informatics

Tomas Bata University in Zlín
Faculty of Applied Informatics
Department of Informatics and Artificial Intelligence

Academic year: 2024/2025

ASSIGNMENT OF DIPLOMA THESIS

(project, art work, art performance)

Name and surname:	Aderiye Oluwasijibomi Tayo
Personal number:	A23603
Study programme:	N0613A140023 Information Technologies
Specialization:	Software Engineering
Type of Study:	Full-time
Work topic:	Analysis of Chunking Strategies for LLM Applications and Proposal of a New Strategy

Theses guidelines

1. Review the literature about Chunking Strategies in the context of building LLM-related applications and related concepts, such as Generative AI, RAG pattern, Embeddings models, vector databases and vector storage.
2. Analyze the technical requirements, capabilities, and limitations of three selected chunking strategies.
3. Develop .NET or Python applications which implement the selected chunking strategies and compare them using metrics such as correctness, relevancy, faithfulness, and semantic similarity.
4. Design a new chunking strategy based on the analysis of existing chunking techniques and implement it by building an application.
5. Compare the proposed chunking strategy against the selected chunking techniques using metrics.
6. Recommend the best practices for chunking strategies in LLM applications.

Form processing of diploma thesis: **printed/electronic**
Language of elaboration: **English**

Recommended resources:

1. ROTHMAN, Denis.. RAG-DRIVEN GENERATIVE AI: Build Custom Retrieval Augmented Generation Pipelines with Llama-index, . . . Deep Lake, and Pinecone. Packt, 2024
2. ASHISH, Kumar. Mastering Text Mining with R: Extract and recognize your text data. Packt, 2016.
3. BAHREE, Amit. Generative AI in Action. Manning, 2024
4. KIMOTHI, Abhinav. A Simple Guide to Retrieval Augmented Generation. Manning, 2024
5. RODRIGUES ANTAO, Tiago. Fast Python, High performance techniques for large datasets. Manning, 2023.
6. GRAINGER, Trey, TURNBULL, Doug, IRWIN, Max. AI-Powered Search. Manning, 2019.
7. BRATANIČ, Tomaž, HANE, Oskar. Knowledge Graph-Enhanced RAG. Manning, 2024.
8. BROUSSEAU, Christopher, SHARP, Matthew. LLMs in Production, From language models to successful products. Manning, 2022.
9. INFANTE, Roberto. AI Apps with LangChain, Manning, 2024.

Supervisors of diploma thesis: **MSc. Luis Antonio Beltran Prieto**
Department of Informatics and Artificial Intelligence

Date of assignment of diploma thesis: **October 27, 2024**

Submission deadline of diploma thesis: **June 2, 2025**

doc. Ing. Jiří Vojtěšek, Ph.D. m.p.
Dean



prof. Mgr. Roman Jašek, Ph.D., DBA m.p.
Head of Department

In Zlín November 17, 2024

MASTER'S THESIS AUTHOR'S DECLARATION

I take cognizance of the fact that

- by submitting my Master's thesis, I give consent to the publication of my thesis in accordance with the Act No. 111/1998 Coll., as amended, regardless of the result of the thesis defence;
- the Master's thesis will be stored in electronic form in the university information system and will be available for consultation;
- one copy of the Master's thesis in hard copy form will be kept and stored by Tomas Bata University in Zlín;
- my Master's thesis is fully subject to the Act No. 121/2000 Coll. on Copyright, on Rights Related to Copyright and on Amendment to Certain Acts (Copyright Act), as amended, particularly to Section 35 Paragraph 3;
- according to Section 60 Paragraph 1 of the Copyright Act, Tomas Bata University in Zlín has the right to conclude a licence agreement on the utilization of a school work as is specified in Section 12 Paragraph 4 of the Copyright Act;
- according to Section 60 Paragraphs 2 and 3, I may utilize my school work – Master's thesis – or grant a licence to utilize it only with the prior written consent of Tomas Bata University in Zlín, which in such a case is entitled to require me to pay a reasonable contribution in order to cover the costs incurred by Tomas Bata University in Zlín for the preparation of the work (up to their full amount);
- if software provided by Tomas Bata University in Zlín or other entities was used for the preparation of the Master's thesis only for learning or research purposes (i.e. for non-commercial use), the conclusions of the Master's thesis cannot be used for commercial purposes;
- if the output of the Master's thesis is any type of software product, the source codes and/or files of which the project is comprised are also considered part of the thesis; failure to submit this part may result in a failure to defend the thesis.

I declare that

- I have worked on my Master's thesis independently and properly cited the literature used; in case of publication of the conclusions, I shall be listed as co-author;
- the submitted hard copy version of the Master's thesis and the electronic version uploaded to the IS/STAG are identical in content.

In Zlín on

.....

author's signature

ABSTRAKT

Velké jazykové modely (LLM) se staly nedílnou součástí moderních aplikací zpracování přirozeného jazyka (NLP), avšak jejich efektivní využití v úlohách, jako je generování doplněné vyhledávání (RAG) a sémantické vyhledávání s využitím embeddingů, závisí na způsobu, jakým je vstupní text rozčleněn do zvládnutelných jednotek. Tato práce zkoumá důležitost strategií segmentace textu při zachování sémantické koherence a současném dodržení limitů tokenů i výpočetních omezení. Srovnávací analýza tří běžně používaných metod – segmentace založené na větách, rekurzivní segmentace a sémantické segmentace – zdůrazňuje související kompromisy. Segmentace založená na větách zachovává gramatické hranice a srozumitelnost, což je užitečné pro krátké kontexty, avšak v delších dokumentech má sklon tříštit kontext. Rekurzivní segmentace zlepšuje kontinuitu rozdělením textu na přirozených hranicích (např. odstavcích či sekcích) s překryvy, čímž zachovává širší kontext a poskytuje přesnější odpovědi ve scénářích typu otázka-odpověď. Sémantická segmentace seskupuje obsah podle významu (na základě podobnosti embeddingů), aby každý segment byl tematicky koherentní, což zlepšuje relevanci vyhledaných informací; tento postup však přináší vyšší výpočetní zátěž a může neúmyslně rozdělit logicky propojený obsah kvůli přísným prahům podobnosti.

Na základě těchto poznatků práce navrhuje novou strategii nazvanou „Markdown-Aware Semantic Chunking“ (MASC). MASC je hybridní přístup, který využívá strukturu dokumentu (s využitím formátovacích prvků Markdownu, jako jsou nadpisy a seznamy) v kombinaci se sémantickou analýzou využívající umělou inteligenci k tvorbě segmentů. Integrací strukturních hranic s mírami sémantické podobnosti MASC zachovává vnitřní logické členění dokumentu a současně udržuje soudržný kontext uvnitř každého segmentu. Tato strategie optimalizuje rovnováhu mezi sémantickou věrností a efektivitou, čímž překonává tradiční metody.

Pro vyhodnocení jednotlivých metod byl použit dotazově-odpovědní systém založený na vyhledávání, který posuzuje každou metodu segmentace podle metrik správnosti (přesnost odpovědi generovaných LLM), relevance (soulad s dotazem), věrnosti (dodržení obsahu zdrojového textu bez halucinací) a sémantické podobnosti (významový překryv mezi odpověďmi a zdrojem). MASC dosahuje nejvyšších hodnot ve všech těchto metrikách a generuje odpovědi, které jsou přesnější, relevantnější a věrnější než při použití segmentace založené na větách, rekurzivní nebo standardní sémantické segmentace. Za pozornost stojí, že díky dělení respektujícím formát, které udržuje související informace pohromadě,

poskytuje MASC modelu LLM úplnější kontext, což následně vede k tomu, že odpovědi jsou přesnější a zároveň kontextově přiléhavé.

Tato zjištění podtrhují význam pokročilých strategií segmentace textu pro reálné systémy založené na velkých jazykových modelech. Řešení využívající LLM a techniku MASC dokáže lépe uchovat kontext i význam, což vede ke zlepšení výkonu v aplikacích, jako je zodpovídání dotazů nad dokumenty, sumarizace nebo vyhledávání znalostí. V praktické rovině nabízí sémantická segmentace zohledňující strukturu spolehlivější a informačně bohatší výstupy LLM – s menším počtem opomenutí a nižším rizikem dezinformací – což je obzvláště přínosné v nasazeních s vysokými nároky nebo ve velkém měřítku. Navrhovaná metoda tedy představuje významný krok vpřed v překlenování propasti mezi surovými schopnostmi LLM a potřebami produkčních AI systémů, dokládá, že segmentace textu není pouze krokem předzpracování, nýbrž kritickým faktorem celkové efektivity řešení využívajících LLM.

ABSTRACT

Large Language Models (LLMs) have become integral to modern NLP applications, but their effective use in tasks like retrieval-augmented generation (RAG) and semantic search with embeddings hinges on how input text is **chunked** into manageable units. This thesis investigates the importance of chunking strategies in preserving semantic coherence while respecting token limits and computational constraints. A comparative analysis of three prevalent methods – **sentence-based chunking**, **recursive chunking**, and **semantic chunking** – highlights the trade-offs involved. Sentence-based chunking preserves grammatical boundaries and clarity, making it useful for short contexts, but it tends to fragment context in longer documents. Recursive chunking improves continuity by splitting text at natural boundaries (e.g. paragraphs or sections) with overlaps, thereby retaining broader context and yielding more accurate answers in question-answering scenarios. Semantic chunking groups content by meaning (using embedding similarity) to ensure each chunk is topically coherent, improving the relevance of retrieved information; however, this method incurs higher computational overhead and can inadvertently split logically connected content due to strict similarity thresholds.

Building on these insights, the thesis proposes a novel **Markdown-Aware Semantic Chunking (MASC)** strategy. MASC is a hybrid approach that leverages document structure (using Markdown formatting cues like headings and lists) in combination with AI-driven semantic analysis to form chunks. By integrating structural boundaries with semantic similarity measures, MASC preserves the document's inherent logical groupings while maintaining cohesive context within each segment. This strategy optimizes the balance between semantic fidelity and efficiency, outperforming the traditional methods. The evaluation framework employs a retrieval-based QA pipeline to assess each chunking method, using metrics of **correctness** (accuracy of LLM-generated answers), **relevancy** (alignment with the query), **faithfulness** (adherence to source content without hallucinations), and **semantic similarity** (meaning overlap between answers and source). MASC achieves the highest scores across all these metrics, producing more accurate, relevant, and faithful responses than the sentence-based, recursive, or standard semantic chunkers. Notably, by keeping related information together through format-aware splitting, MASC provides the LLM with more complete context, which in turn yields answers that are both precise and contextually appropriate.

These findings underscore the significance of advanced chunking strategies for real-world LLM systems. An LLM pipeline employing the MASC technique can better preserve context and meaning, leading to improved performance in applications such as document question-answering, summarization, and knowledge-based retrieval. In practical terms, structure-aware semantic chunking offers more reliable and informative LLM outputs – with fewer omissions and reduced risk of misinformation – especially valuable in high-stakes or large-scale deployments. The proposed method thus represents a meaningful step forward in bridging the gap between raw LLM capabilities and the needs of production AI systems, demonstrating that chunking is not merely a preprocessing step but a critical factor in the overall effectiveness of LLM-driven solutions.

ACKNOWLEDGMENT

I would like to thank **God** for His guidance throughout this work. I am deeply grateful to my **mother**, **sister**, and **brother** for their unwavering support, and I extend my heartfelt thanks to my late **father**, whose mentorship and inspiration have been instrumental in this journey.

CONTENTS

INTRODUCTION	15
I THEORETICAL PART	18
1 LITERATURE REVIEW	19
1.1 CHUNKING STRATEGIES: DEFINITION AND IMPORTANCE	20
1.2 RAG PATTERNS AND GENERATIVE AI?	21
1.3 EMBEDDING MODELS	22
1.4 VECTOR DATABASES AND VECTOR STORAGE TECHNIQUES	23
1.5 SEMANTIC SIMILARITY	24
1.6 MOTIVATION AND NEED FOR CHUNKING STRATEGIES	25
II PRACTICAL PART	27
2 ANALYSIS OF CHUNKING STRATEGIES	29
2.1 SENTENCE-BASED CHUNKING	29
2.2 RECURSIVE CHUNKING	31
2.3 SEMANTIC-BASED CHUNKING	32
2.4 MARKDOWN-AWARE SEMANTIC CHUNKING (MASC)	33
3 IMPLEMENTATION OF FRONTEND APPLICATION FOR VISUALIZING CHUNKING STRATEGIES	38
3.1 SYSTEM DESIGN	38
3.2 STRATEGY SELECTION AND API INTEGRATION	40
3.3 DISPLAYING THE CHUNKED DATA	42
4 COMPARISON OF CHUNKING STRATEGIES	45
4.1 CODE EXPLANATION: CHUNKING STRATEGY EVALUATION PIPELINE	46
4.2 RECURSIVE CHUNKER Vs TEXT OVERLAP CHUNKER	50
4.3 SEMANTIC CHUNKER Vs CUSTOM MASC CHUNKER	53
CONCLUSION	56
BIBLIOGRAPHY	59
LIST OF FIGURES	61
LIST OF TABLES	62
LIST OF ABBREVIATIONS	63

INTRODUCTION

In recent years, the field of artificial intelligence has experienced rapid advancements, with LLMs emerging as foundational technologies in numerous applications. These models, exemplified by architectures like OpenAI's GPT series and Google's BERT, have demonstrated unprecedented capabilities in understanding and generating human-like text. Their adoption spans diverse domains, including natural language processing (NLP), automated decision-making systems, and conversational Artificial Intelligence (AI). However, leveraging the full potential of LLMs in real-world scenarios requires preprocessing input data into manageable units commonly referred to as "chunks." The method used to divide this data plays a pivotal role in determining model performance, computational efficiency, and the interpretability of the generated outputs.

Existing literature underscores the importance of semantic-aware chunking in enhancing performance for tasks such as summarization, translation, and question answering. Research has shown that preserving semantic coherence within chunks significantly improves the relevance and accuracy of model outputs. For example, chunking methods that respect sentence boundaries or thematic continuity often produce outputs that are more aligned with human expectations. Despite these advantages, achieving semantic coherence often requires computationally intensive processes, such as dependency parsing, embedding-based similarity calculations, or contextual analysis. These processes, while effective, introduce latency and pose challenges to scalability, particularly in time-sensitive or high-throughput applications.

The challenge, therefore, lies in developing chunking strategies that strike a balance between maintaining semantic integrity and ensuring computational efficiency. Traditional approaches often operate under fixed rules, such as character or token limits, which may fail to account for the contextual nuances of the input data. On the other hand, more sophisticated techniques, which adapt based on semantic features, risk becoming bottlenecks in real-time systems due to their computational overhead.

This thesis seeks to address this critical gap by providing a comprehensive analysis of existing chunking methodologies and their applicability across various tasks. Through a detailed review, we identify the strengths and limitations of popular strategies, examining their performance in different contexts and under varying constraints. For instance, while token-based chunking methods are straightforward and computationally efficient, they may

inadvertently split semantically connected ideas, leading to degraded model outputs. Conversely, semantic-aware techniques, though more accurate, often struggle with scalability in environments requiring low latency and high throughput.

Building upon these insights, we propose a novel chunking strategy that optimizes the trade-off between semantic coherence and computational efficiency. Our approach introduces adaptive mechanisms capable of dynamically adjusting chunk sizes and boundaries based on task-specific requirements and the characteristics of the underlying model. By integrating lightweight semantic analysis with rule-based adjustments, the proposed strategy ensures that chunks remain contextually meaningful without incurring significant computational overhead. For instance, the strategy may leverage statistical patterns or low-cost heuristics to approximate semantic boundaries, ensuring a pragmatic yet effective solution.

The implications of this research are far-reaching. By enhancing chunking strategies, we aim to improve the overall utility of LLMs in a wide range of applications, from academic research and content creation to industrial deployments and real-time systems. Furthermore, our approach is designed to be flexible, allowing it to accommodate advancements in LLM architectures and the growing diversity of NLP tasks. As LLMs continue to evolve, the need for efficient and semantically aware preprocessing techniques will remain critical. This work aspires to contribute to that need, bridging the gap between theoretical advancements and practical implementations in the dynamic landscape of AI-powered solutions.

In conclusion, while existing chunking strategies have laid a solid foundation for leveraging LLMs effectively, there remains significant room for improvement in balancing semantic fidelity with computational demands. By proposing an innovative, adaptive chunking framework, this thesis aims to address these challenges, paving the way for more efficient and impactful use of LLMs across diverse scenarios.

OBJECTIVES

1. **Review Literature:** Conduct a thorough review of chunking strategies in the context of building LLM-related applications and related concepts such as Generative AI, Retrieval-Augmented Generation (RAG) patterns, embeddings models, vector databases, and vector storage.
2. **Analyze Strategies:** Examine the technical requirements, capabilities, and limitations of three selected chunking strategies, focusing on their applicability to real-world tasks.
3. **Develop Applications:** Create .NET or Python applications that implement the selected chunking strategies, enabling their comparison using metrics such as correctness, relevancy, faithfulness, and semantic similarity.
4. **Propose a New Strategy:** Design and implement a novel chunking strategy based on insights from the analysis of existing techniques, incorporating adaptive mechanisms to optimize semantic coherence and computational efficiency.
5. **Compare Techniques:** Evaluate the proposed chunking strategy against existing ones using quantitative metrics to assess performance improvements.
6. **Recommend Best Practices:** Develop guidelines for selecting and implementing chunking strategies in LLM applications to maximize effectiveness across various use cases.

I THEORETICAL PART

1 LITERATURE REVIEW

The rapid advancement of LLMs has ushered in a new era of possibilities for NLP applications. These applications span a wide array of domains, including document summarization, question answering, conversational AI, knowledge retrieval, and content generation. A fundamental aspect of optimizing LLM-based applications lies in the development and application of chunking strategies—methods that govern how information is segmented into manageable units for processing and storage. Chunking is vital for several reasons like Token Management, Retrieval Accuracy in RAG, Context Preservation and Efficiency[14].

Chunking strategies address the challenge of handling vast datasets by dividing them into smaller, more coherent pieces, enabling efficient processing while retaining semantic integrity. These strategies are crucial in managing the limitations of LLMs, such as token constraints and context size restrictions, and in optimizing the interaction between LLMs and auxiliary technologies like vector databases and embeddings models.

This literature review explores the diverse chunking methodologies and their applications in LLM-related workflows. Focus is placed on how these strategies integrate with and enhance modern technological frameworks, including:

Generative AI: Understanding how chunking contributes to the generation of coherent, contextually relevant outputs by maintaining a balance between input granularity and contextual comprehensiveness.

RAG Patterns: Exploring the role of chunking in improving retrieval accuracy and contextual relevance in RAG workflows, where retrieved chunks of data serve as prompts for generating informed responses.

Embeddings Models: Investigating the role of chunking in creating and managing vectorized representations of text, which are used for similarity matching and semantic search.

Vector Storage and Techniques: Analyzing how chunking strategies impact the efficiency, accuracy, and optimization of vector-based storage and retrieval systems. This includes exploring their role in knowledge-intensive applications and how different chunking methods enhance data retrieval speed and relevance.

1.1 Chunking Strategies: Definition and Importance

Chunking involves dividing text into manageable units or “chunks” for efficient processing. This segmentation is essential for tasks like semantic search, information retrieval, and generative AI applications. Each chunk retains context and semantic integrity to ensure coherent results [1].

Various chunking techniques cater to specific needs based on the structure of the text and application requirements, for example:

Fixed-Length Chunking:

Fixed-length chunking divides text into equal-sized segments based on tokens, words, or characters. While efficient for processing, it may break context at boundaries. Variants like Sliding Window Chunking help retain context by overlapping chunks. Here are some examples of Fixed-Length Chunking strategies;

- **Document-Based Chunking:** Considers the entire document as a single chunk. This approach is effective for maintaining overall context but may not be feasible for large texts due to memory constraints. Document chunking involves breaking down a large text into smaller, manageable pieces or "chunks" that are easier to process and analyze by computational models. This is crucial in scenarios where the answer to a query might span different sections of a document or when handling large datasets that exceed the processing capacity of standard models [7].
- **Fixed-Length Chunking:** Divides text into uniform sizes based on tokens, words, or characters. It is computationally efficient but may cut off meaningful context at boundaries.
- **Sentence-Based Chunking:** Segments text by sentences, preserving grammatical and contextual integrity. It is ideal for conversational models but can be inefficient for longer texts.
- **Paragraph-Based Chunking:** Group text by paragraphs, maintain thematic context. Suitable for structured documents but may lose granularity for fine-tuned tasks.

- **Sliding Window Chunking:** Uses overlapping windows to segment text, reducing information loss at chunk boundaries. It ensures better context retention but increases memory and processing costs.

Semantic Chunking:

Focuses on grouping text by meaning rather than structure. This ensures semantic coherence but increases computational overhead as it requires deep language understanding.

Chunking is crucial in LLMs for efficient processing, accurate responses, and optimized performance. Since models like GPT have token limits, breaking larger texts into smaller segments ensures they fit within these constraints without losing important information. Context management also benefits from chunking, as smaller, cohesive chunks help maintain understanding and improve generation accuracy. Additionally, efficient chunking reduces computational overhead and enhances retrieval relevance, making it essential for optimizing performance in large-scale language processing tasks[14].

1.2 RAG patterns and Generative AI?

Generative AI describes algorithms (such as GPT-4, BERT, LLaMA, Claude) that can be used to create new content, including audio, code, images, text, simulations, and videos [2].

Even the most sophisticated generative models are confined to responding within the boundaries of the data they were trained on. They are unable to accurately address queries involving information outside of their knowledge base, such as new information. This lack of awareness about their limitations often results in flawed or irrelevant outputs, sometimes described as errors, biases, or simply unreliable information.

Chunking strategies in Generative AI significantly impact the quality of generated responses by balancing input size (how small or large a chunk is) and comprehensive context (how much useful context is preserved). Maintaining a balance where each chunk provides enough self-contained meaning while also contributing to the global coherence of generated response is important.

RAG is a framework that addresses this limitation by combining retrieval-based approaches with generative models. It retrieves relevant data from external sources in real time and use this data to generate more accurate and contextually relevant responses. Generative AI

models integrated with RAG retrievers are revolutionizing the field with their unprecedented efficiency and power. One of the key strengths of RAG is its adaptability. It can be seamlessly applied to any type of data, be it text, images, or audio. This versatility makes RAG ecosystems a reliable and efficient tool for enhancing generative AI capabilities [3].

In RAG workflows, chunking directly affects retrieval efficiency and the relevance of generated responses by determining what information is available as context. Improving retrieval accuracy and contextual relevance, ensuring the right amount of information is retrieved for generating high quality response is a key factor when considering what chunking method to be used.

1.3 Embedding Models

Embeddings are numerical representations of real-world objects that machine learning (ML) and AI systems use to understand complex knowledge domains like humans do [4]. Embeddings convert real-world objects into complex mathematical representations that capture inherent properties and relationships between real-world data [4].

Embedding techniques, such as word, sentence, or document embeddings, map textual data to dense vector representations that encode semantic structure. The chunking scheme has an important effect on how the embeddings can effectively be obtained and exploited to accomplish tasks such as semantic indexing or pattern matching.

Embedding models have revolutionized machine learning by enabling the transformation of complex data into lower-dimensional representations that are easier to process. These models have been particularly impactful in fields such as NLP, computer vision, and recommendation systems. Embedding refers to mapping high-dimensional data (e.g., text, images) into dense, lower-dimensional vectors that preserve semantic relationships[8]. Examples of embedding models; GPT,BERT,text-embedding-ada-002.

Chunking plays a crucial role in shaping how embedding models process and represent text, directly impacting granularity, contextuality, and overall search accuracy. Smaller chunks generate highly specific embeddings that capture fine semantic details but may lose broader context, while larger chunks provide richer meaning representation but risk including extraneous information. Embedding models perform best when they incorporate coherent contextual data, and effective chunking ensures that text snippets remain meaningful while minimizing noise. Semantic chunking, which groups text around natural boundaries like

sentences, paragraphs, or topics, enhances semantic search accuracy and similarity matching. However, the challenge lies in balancing small, context-specific chunks for precise embeddings and larger, context-rich chunks for a more comprehensive understanding, ultimately optimizing retrieval performance.

1.4 Vector Databases and Vector Storage Techniques

A vector database indexes and stores vector embeddings for fast retrieval and similarity search, with capabilities like CRUD (Create-Read-Update-Delete) operations, metadata filtering, horizontal scaling, and serverless [5]. The way text is chunked affects how well vector data is stored, indexed, and later retrieved.

Vector storage techniques are critical for ensuring efficient and fast data retrieval. The way text is chunked affects the overall performance of storage optimization and the relevance of retrieved data. Optimized chunking helps manage storage requirements, improves indexing, and accelerates retrieval while maintaining semantic relevance.

Vector storage techniques define how high-dimensional vectors (e.g., 768-dimensional sentence embeddings) are stored and accessed within a vector database. These techniques impact performance, scalability, fault tolerance, and query latency. Key storage strategies include:

- **Flat File storage with Brute-Force Search:** Vectors are stored sequentially in a file or memory block, and similarity searches are performed using brute-force comparisons across all stored vectors. While accurate, this approach is computationally expensive and doesn't scale well with millions of vectors.
- **Hierarchical Storage:** Vectors are stored across memory (RAM), SSD, and cold storage based on frequency of access. Frequently accessed vectors remain in memory for fast retrieval, while less-used vectors are moved to disk or object storage. This technique balances speed and cost-efficiency.
- **Metadata-Aware Storage:** Vectors are stored alongside metadata (e.g., document title, tags, language). The storage engine supports filtering during search by associating each vector with a key-value store, enabling hybrid search (semantic + keyword).

- **Memory-Mapped Files (MMAP):** Used in high-throughput environments where large vector indices are loaded into memory as mapped files, allowing faster access without loading entire datasets into RAM.

Chunking significantly impacts the efficiency of vector databases, influencing storage, search precision, and indexing performance. Smaller chunks increase the number of database entries but improve retrieval speed by making each vector more precise and closely aligned with queries. In contrast, larger chunks reduce the total entries but may slow searches due to added noise and irrelevant data, affecting retrieval accuracy. Vector search relies on finding the closest match to a query vector, and smaller, more specialized chunks enhance precision, whereas larger chunks introduce variability, potentially broadening results at the cost of relevance. Proper chunking also optimizes indexing by structuring data coherently, reducing the time required for updates and searches. However, poorly aligned chunks can create inefficient indices, leading to slower retrieval and less relevant results. Some examples of vector databases are: Pinecone, Chroma, Azure Cognitive search.

1.5 Semantic Similarity

Semantic similarity measures how closely the meanings of two pieces of text align, regardless of their surface-level word choices. It is a core principle behind vector search and RAG.

In vector databases, embeddings generated from text preserve semantic meaning in high-dimensional space. Queries and documents that are semantically similar will have embeddings that lie closer together in the vector space. The chunking strategy used significantly affects this relationship:

- **Fine-grained (smaller) chunks** preserve local context and are better for pinpoint semantic retrieval (e.g., answering specific questions).
- **Coarse-grained (larger) chunks** capture broader themes but can dilute specific meaning, leading to less accurate semantic similarity matches.

In RAG pipelines, semantic similarity directly determines which context chunks from a database are retrieved and passed to the language model for answer generation. Therefore, chunk size, coherence, and structure directly influence the quality of generated answers.

Proper semantic similarity alignment also impacts:

- Context injection in LLMs
- FAQ systems
- Document summarization
- Personalized recommendations

Thus, designing vector storage and chunking strategies with semantic similarity in mind is crucial for high-performance search and generation systems.

1.6 Motivation and Need for Chunking Strategies

After exploring the mechanics of vector storage and semantic similarity, it becomes clear that chunking is not a trivial preprocessing step—it is a foundational strategy that influences the system's overall effectiveness.

Key motivations for thoughtful chunking include:

- **Search Optimization:** Well-structured chunks increase precision in vector similarity searches.
- **Cost and Resource Management:** Balanced chunking reduces operational costs in cloud-based vector DBs.
- **Scalability:** Chunking affects how data can be partitioned, indexed, and parallelized across nodes. It enables efficient handling of millions of documents in real-time systems.
- **Relevance and Context Preservation:** Maintaining coherent context within a chunk ensures higher semantic fidelity. Helps LLMs generate grounded, contextually accurate answers during inference.
- **Support for Downstream Tasks (like RAG):** Chunking impacts how LLMs use retrieved context to answer queries.

In conclusion, chunking is both an art and a science. The choice of chunk size, method (sliding window, sentence-based, topic-based), and overlap percentage should be aligned with the specific use case—whether it's dense retrieval, hybrid search, or LLM integration. Optimizing chunking unlocks higher precision, lower latency, and better end-user experiences in AI-driven systems.

II PRACTICAL PART

2 ANALYSIS OF CHUNKING STRATEGIES

The analysis phase of this research focuses on systematically comparing different chunking strategies to evaluate their effectiveness in LLM applications. This involves assessing each method based on key factors such as context retention, implementation using specific python tools or packages, limitation, application of strategies and retrieval accuracy.

2.1 Sentence-Based Chunking

Sentence-based chunking is a fundamental text processing technique that divides a given textual corpus into discrete sentence-level units. This approach facilitates a structured representation of information, enhancing tasks such as NLP, information retrieval, and machine learning applications. By segmenting text into sentences, sentence-based chunking enables efficient linguistic analysis and contributes to improved contextual understanding.

Text preprocessing is an essential step in NLP that involves cleaning and transforming unstructured text data to prepare it for analysis [6]. Text processing in NLP often requires meaningful segmentation to ensure that computational models accurately interpret context and semantics. Sentence chunking is grounded in the principles of syntactic and grammatical structuring, where each sentence functions as an independent linguistic unit. Leveraging linguistic rules and machine learning models, modern sentence chunking techniques account for punctuation marks, language-specific syntactic structures, and named entity recognition (NER) cues.

Sentence-based chunking enables numerous NLP applications. By utilizing tools like spaCy [17], this technique enhances text segmentation, improving downstream tasks such as summarization, translation, and sentiment analysis.

Implementation using spaCy

One of the most effective ways to achieve sentence-based chunking is by using NLP libraries such as spaCy. SpaCy provides a robust framework for sentence tokenization, leveraging pre-trained language models to accurately segment text. The implementation begins by loading a pre-trained language model, such as `en_core_web_sm`, which facilitates sentence detection. Once the model is loaded, the input text is processed through spaCy's NLP pipeline, where it undergoes tokenization. After processing, the document is iterated to

extract individual sentences, ensuring accurate segmentation. Finally, each sentence is stored in a structured format, containing a unique identifier along with the sentence content, making it easy to manage and analyze.

```
1 import spacy
2
3 def sentence_chunking(text: str):
4     """Splits text into sentences using spaCy."""
5     nlp = spacy.load("en_core_web_sm")
6     doc = nlp(text)
7     chunks = []
8     for i, sent in enumerate(doc.sents):
9         chunks.append({
10             "chunk_id": i + 1,
11             "content": sent.text,
12             "num_sentences": 1
13         })
14     return chunks
```

Listing 1 Implemetation of sentence chunking using spaCy.

Limitation of Sentence-Based Chunking

Sentence-Based Chunking is a useful technique for text segmentation, but it has several limitations;

- **Limited Scope:** While it maintains sentence-level meaning, sentence based chunking might not capture broader relationships between sentences that span multiple chunks [9].
- **Potential Redundancy:** Depending on the nature of the text, there could be redundancy across consecutive sentences within a chunk[9].

Sentence-based chunking is widely used in text summarization, where key sentences are extracted to generate concise summaries. In sentiment analysis, it helps evaluate emotions and opinions within different parts of a document. Machine translation benefits from sentence-based segmentation by ensuring accurate syntactic alignment between languages. Additionally, named entity recognition (NER) improves with precise sentence boundaries, enhancing the identification of important entities. In text-to-speech (TTS) systems, proper sentence segmentation ensures natural pauses, leading to more fluid and human-like speech synthesis.

2.2 Recursive Chunking

Recursive chunking is a text segmentation technique that employs either token-based or source-text-based chunking to recursively divide a text into smaller units. In this method, larger chunks are initially segmented using token-based or source-text-based chunking techniques. Then, each of these larger chunks is further subdivided into smaller segments using the same chunking approach. This recursive process continues until the desired level of granularity is achieved or until certain criteria are met. Its drawback is computational inefficiency[10].

While Fixed Size Chunking is easy to implement, it neglects the text structure. Recursive Chunking, on the other hand, divides text into smaller chunks hierarchically and iteratively using a set of separators. If the initial attempt at splitting the text doesn't produce chunks of the desired size or structure, the method recursively calls itself on the resulting chunks with a different separator or criterion until the desired chunk size or structure is achieved. This means that while the chunks are not going to be exactly the same size, they will still “aspire” to be of a similar size.

Implementation of recursive chunking using Langchain framework;

LangChain [18] is an open-source framework for building applications based on LLMs that provides robust support for chunking strategies, which are essential for processing and retrieving large text documents efficiently. It includes built-in text splitters that enable sentence-based, fixed-size, and semantic chunking methods.

Langchain offers RecursiveCharacterTextSplitter class which is a text chunking tool designed to split large text into smaller, manageable pieces while maintaining logical coherence. Unlike simple character-based splitting, it recursively divides text using hierarchical approach, prioritizing sentence and paragraph before falling back to character - level splitting if necessary.

```
1 from langchain.text_splitter import
  RecursiveCharacterTextSplitter
2 def recursive_chunking_method(text, chunk_size=500,
  chunk_overlap=50):
3
4     # Create a text splitter
5     splitter =
  RecursiveCharacterTextSplitter(chunk_size=chunk_size,
  chunk_overlap=chunk_overlap)
```

```
6
7     # Split the text using the splitter
8     chunks = splitter.split_text(text)
9
10    return chunks
```

Listing 2 Implementation of Recursive Chunking using LangChain

Limitation for Recursive Chunking:

While recursive chunking improves text segmentation by dynamically adjusting chunk sizes, it still has certain limitations that can impact efficiency, context retention, and processing accuracy.

- **Computational Overhead** – The recursive approach requires multiple passes over the text, making it more computationally expensive compared to simple fixed-size chunking.
- **Inefficiency with Highly Structured Text** – Documents with structured formatting, such as tables, code blocks, or mathematical equations, may not be split optimally, leading to incomplete or fragmented chunks.
- **Potential Redundancy** – To ensure context retention, some overlap between chunks may be introduced, leading to redundant processing and storage when working with embeddings or language models.

Recursive chunking preserves the document hierarchy by maintaining the logical flow of structured documents, starting with high-level divisions before breaking them down further. It effectively balances context and token limits, ensuring that chunks are not too large while still retaining sufficient information for meaningful processing. Additionally, this method is highly flexible and scalable, making it suitable for various document types, from simple articles to complex technical manuals, without losing coherence or readability.

2.3 Semantic-Based Chunking

Semantic Chunking considers the relationships within the text. It divides the text into meaningful, semantically complete chunks. This approach ensures the information's integrity during retrieval, leading to a more accurate and contextually appropriate outcome. It is slower compared to the previous chunking strategy[12].

Semantic chunking involves taking the embeddings of every sentence in the document, comparing the similarity of all sentences with each other, and then grouping sentences with the most similar embeddings together[12].

Implementation of Semantic Chunking using LangChain Framework

One of LangChain advanced text splitters in the **SemanticChunker** [19] which enables text segmentation based on meaning rather than fixed length constraints. This can also split large documents into smaller, semantically cohesive units. Unlike character-based chunking, this method ensures that each segment contains complete ideas rather than arbitrary text portions.

```
1 from langchain.text_splitter import SemanticChunker
2 from langchain.embeddings.openai import OpenAIEmbeddings
3
4 def semantic_chunking_method(text, chunk_size=500,
5   chunk_overlap=50, model="text-embedding-ada-002"):
6     # Create an embedding model
7     embeddings = OpenAIEmbeddings(model=model)
8
9     # Create a semantic chunker
10    splitter = SemanticChunker(embedding_model=embeddings,
11    chunk_size=chunk_size, chunk_overlap=chunk_overlap)
12
13    # Split the text using the chunker
14    chunks = splitter.split_text(text)
15
16    return chunks
```

Listing 3 Implementation of Semantic Chunking using LangChain

Limitation of Semantic-Based Chunking;

While semantic-based chunking offers improved contextual understanding, it also comes with several limitations that can impact its accuracy and efficiency:

- **Complex Implementation:** Requires advanced NLP techniques and custom logic, making it harder to implement than simpler chunking methods[13].
- **Variable chunk sizes:** Similar to other semantic-based methods, the chunk size can vary greatly, leading to issues with token limits or computational inefficiencies[13].

2.4 Markdown-Aware Semantic Chunking (MASC)

MASC is a hybrid chunking strategy that leverages document structure and AI-driven semantic refinement. This method first converts PDFs into Markdown format using **LlamaParse** [20] to retain headers and structural elements. It then applies a header-based splitting approach using LangChain's Markdown text splitter. Finally, an LLM analyzes the

generated chunks for similarity and refines the segmentation by merging semantically related chunks.

This approach maintains structural coherence while ensuring that the text is divided into meaningful, contextually relevant segments. It is particularly useful for processing long-form documents with hierarchical structures, such as research papers, technical manuals, and reports.

Implementation of MASC using LangChain framework and OpenAI

Using **LlamaParse** to convert the text extracted from the pdf uploaded to markdown (a lightweight markup language for creating formatted text using a plain-text editor)

```
1  from llama_parse import LlamaParse
2  import os
3  import re
4
5  def convert_to_markdown_return_string(pdf_path)-> str:
6      parser = LlamaParse(api_key=os.getenv("LLAMA_API_KEY"),
7                          result_type='markdown')
8      documents = parser.load_data(pdf_path)
9
10     # Use a list to accumulate the markdown content instead
    of writing to a file
11     markdown_content = []
12
13     for doc in documents:
14         # Extract the text content from the Document object
15         text = doc.text
16
17         # Split text into lines to process headers
18         lines = text.split('\n')
19
20         # Flag to track if the previous NON-BLANK line was a
    header
21         prev_header_line_index = None
22
23         for i, line in enumerate(lines):
24             stripped_line = line.strip() # Remove
    surrounding spaces
25
26             if stripped_line.startswith('#'): # Check if the
    line is a header
27                 header_content =
                    stripped_line.lstrip('#').strip()
```

```

28         # Check if the header contains ONLY a number
        (e.g., "2.1", "1.10") or a Roman numeral (e.g., "(ii)",
        "(IV)")
29         if re.match(r"^\d+(\.\d+)*$", header_content)
        or re.match(r"^\([ivxlcdm]+\)$", header_content,
        re.IGNORECASE):
30             lines[i] = '# ' + header_content #
        Convert to # subheading
31             elif prev_header_line_index is not None:
32                 lines[i] = '# ' +
        stripped_line.lstrip('#').strip() # Keep it as #
33             else:
34                 lines[i] = '## ' +
        stripped_line.lstrip('#').strip() # First header becomes ##
35
36             prev_header_line_index = i # Update index of
        last header line
37             elif stripped_line: # Reset only if it's a non-
        blank, non-header line
38                 prev_header_line_index = None
39
40         # Append the processed content to the
        markdown_content list
41         markdown_content.append("\n".join(lines))
42
43     # Return the final markdown as a string
44     return "\n\n".join(markdown_content)

```

Listing 4 Converting pdf to Markdown format using LlamaParse

After the successful conversion from PDF to markdown format, LangChain framework is used to split and chunk based on headers. Refer to the code block below

```

1  from langchain_text_splitters import
    MarkdownHeaderTextSplitter
2
3  def chunk_markdown_by_header(markdown_text: str):
4      headers_to_split_on = [
5          ("##", "Header 1"),
6      ]
7      markdown_splitter =
        MarkdownHeaderTextSplitter(headers_to_split_on,
        strip_headers=False)
8      md_header_splits =
        markdown_splitter.split_text(markdown_text)
9
10     result = []
11     for split in md_header_splits:
12         # Access the 'metadata' attribute of the Document
        object
13         header_1 = split.metadata.get("Header 1", "") #
        Safely access Header 1
14
15         # Access the 'page_content' attribute of the Document
        object

```

```

16         page_content = getattr(split, "page_content", "") #
    Safe access to page_content
17
18         result.append({
19             # "Header": header_1,
20             "page_content": normalize_text(page_content)
21         })
22
23     return result

```

Listing 5 Chunk Markdown text by header using LangChain Framework

After generating initial chunks using the Markdown header-based splitting in Listing 5, the OpenAI model is used to assess the semantic similarity between consecutive chunks. If two chunks are determined to be semantically related, they are merged into a single chunk. This ensures that closely related information remains grouped together, improving retrieval accuracy and maintaining contextual integrity. The code below shows how the chunks are compared and deemed to be similar or not

```

1  import os
2  import openai
3  from time import sleep
4  from openai.error import RateLimitError
5
6  def are_chunks_related(chunk1, chunk2):
7      """
8      Determines if two chunks are semantically related using
9      GPT.
10     Retries indefinitely if rate limit is hit.
11
12     :param chunk1: First chunk of text.
13     :param chunk2: Second chunk of text.
14     :return: True if related, False otherwise.
15     """
16     openai.api_type = os.getenv("OPENAI_API_TYPE")
17     openai.api_key = os.getenv("OPENAI_API_KEY")
18     openai.api_base = os.getenv("OPENAI_API_BASE")
19     openai.api_version = os.getenv("OPENAI_API_VERSION")
20
21     prompt = f"""
22     Determine if the following two sections of text belong to
23     the same topic.
24     Respond with ONLY "YES" or "NO".
25
26     Section 1: {chunk1}
27     Section 2: {chunk2}
28
29     Answer:
30     """
31
32     messages = [

```

```
31         {"role": "system", "content": "You are an AI that
determines if two sections are related."},
32         {"role": "user", "content": prompt}
33     ]
34
35     wait_time = 2 # Start with a 2-second delay for retries
36
37     while True:
38         try:
39             response = openai.ChatCompletion.create(
40                 engine=os.getenv("TEXT_DEPLOYMENT_ID"),
41                 messages=messages,
42                 max_tokens=2,
43                 temperature=0
44             )
45
46             answer =
response["choices"][0]["message"]["content"].strip().upper()
47             return answer == "YES"
48
49         except RateLimitError:
50             print(f"Rate limit exceeded. Retrying in
{wait_time} seconds...")
51             sleep(wait_time)
52             wait_time = min(wait_time * 2, 60) # Exponential
backoff (max 60s)
```

Listing 6 Semantic chunking using OpenAI

Limitations of MASC:

While this method improves chunking accuracy and preserves structural integrity it has some limitations:

- **Computationally Intensive:** AI-driven refinement requires additional processing time and API calls, making it slower than traditional chunking methods.
- **Dependency on Document Formatting:** Markdown conversion relies on well-structured PDFs. Poorly formatted documents may result in suboptimal chunking.
- **AI Uncertainty:** The accuracy of chunk merging depends on the LLM's responses, which may introduce inconsistencies.

Despite these challenges, Markdown-Aware AI Chunking offers a powerful way to segment structured documents while maintaining semantic coherence.

3 IMPLEMENTATION OF FRONTEND APPLICATION FOR VISUALIZING CHUNKING STRATEGIES

The Angular frontend provides an interface for users to select a chunking strategy and then requests the backend to perform chunking using that strategy. The application enables users to upload documents, select a chunking strategy, trigger the chunking process via a backend API, and view the resulting text chunks in a user-friendly table format. This design ensures a clean separation between chunking logic (handled by the backend) and presentation logic (handled by the frontend).

3.1 System Design

The system is designed to provide users with a seamless and efficient experience for uploading files and processing them through various advanced chunking strategies. It is structured into two primary components: a responsive frontend interface that interacts with users, and a powerful backend API responsible for executing the core processing logic.

Below are screenshots of the user interface and documentation of the backend API routes that power the file upload and chunking process:

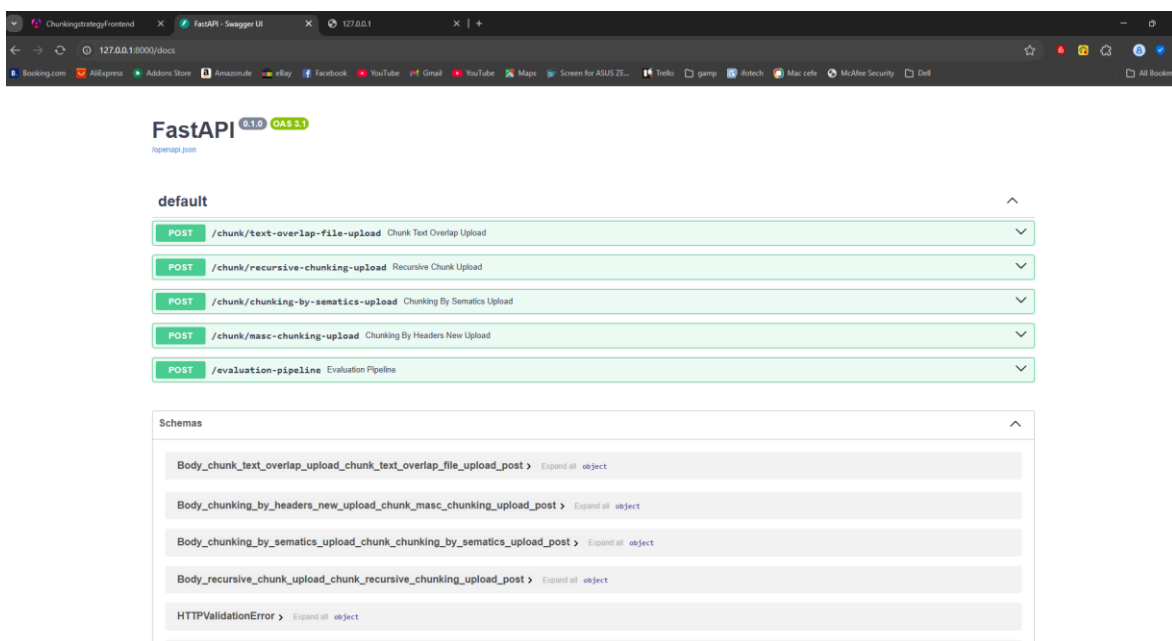


Figure 1 Backend Swagger Page

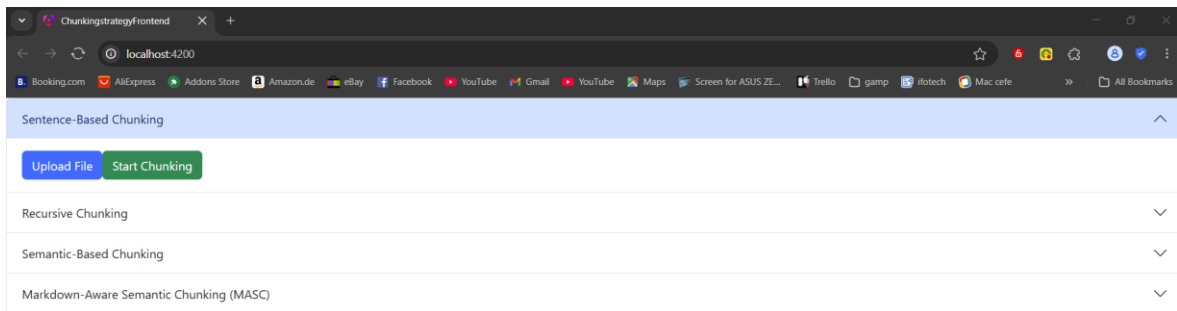


Figure 2 Frontend UI

- Frontend: Angular application with Bootstrap UI.
- Backend API: FastAPI (python framework) receives files and executes chunking strategies.

Interaction Flow

The application provides a smooth and interactive experience for users to upload and process their files. The interaction between the frontend and backend follows a clear sequence, ensuring efficient file handling and immediate feedback for the user. The flow is outlined below:

- User uploads a file.
- User selects a chunking strategy.
- Frontend sends the file to a corresponding backend API endpoint.
- Backend processes the file and returns the chunks.
- Frontend displays the first N chunks (e.g 10) in a dynamic table.

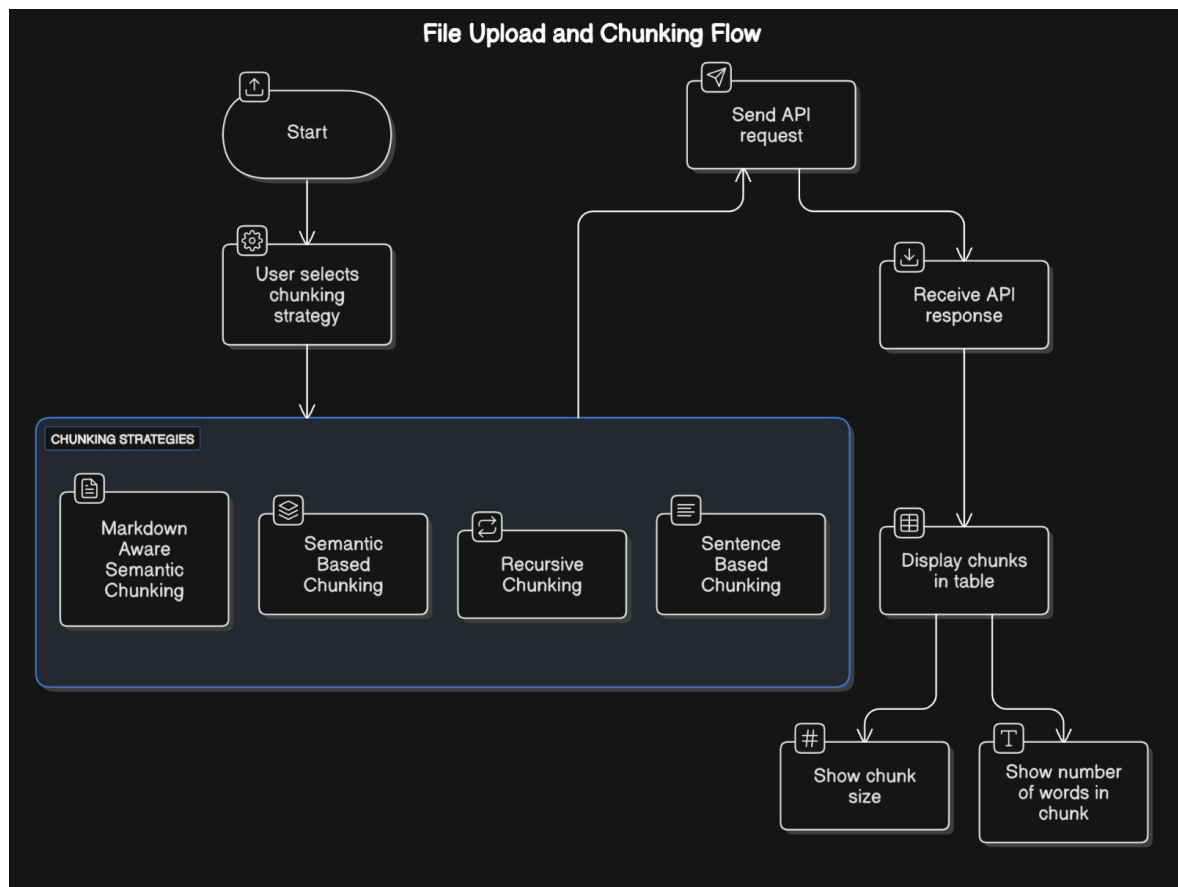


Figure 3 File Upload and Chunking Flow

3.2 Strategy Selection and API integration

The frontend provides an accordion-style menu where users can pick the chunking strategy they want to apply. This functionality is implemented in the `app.component.html` file of the Angular project:

```

1  @for (item of chunkingStrategies; track item) {
2    <div ngbAccordionItem>
3      <h2 ngbAccordionHeader>
4        <button ngbAccordionButton>{{ item.name }}</button>
5      </h2>
6      ...
7    </div>
8  }

```

Listing 7 Accordion-Style Menu

Each accordion section corresponds to one of the four strategies see Figure 4:

- Sentenced-Based Chunking
- Recursive Chunking
- Semantic-Based Chunking
- Markdown-Aware Semantic Chunking (MASC)

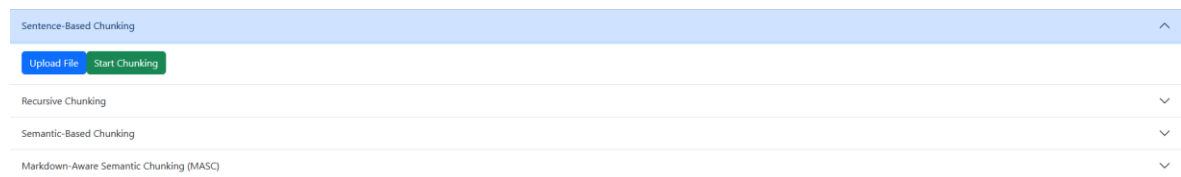


Figure 4 Accordion UI

When a user selects a file as seen in Figure 5 and clicks Start Chunking, the application calls specific methods depending on the selected strategy, such as;

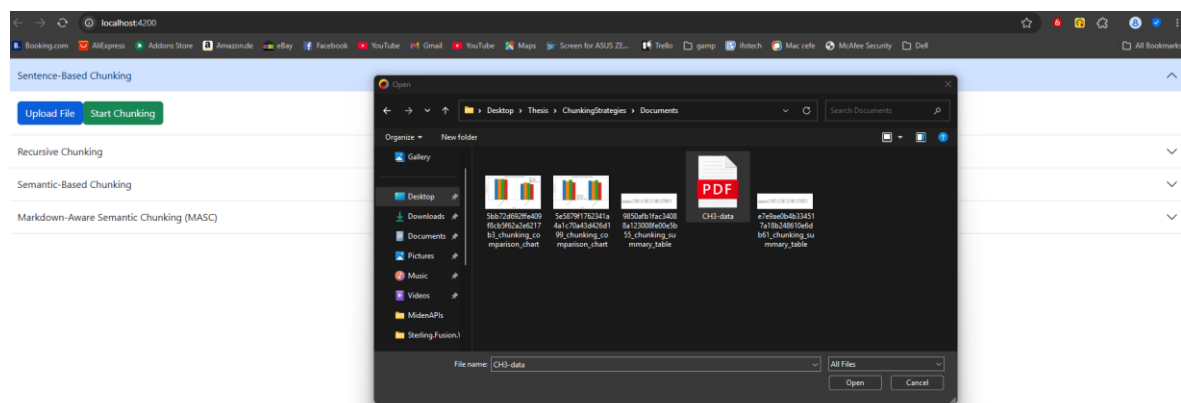


Figure 5 User selects Document to chunk

```
9 sentenceBasedChunking()  
10 recursiveChunking()  
11 semanticChunking()  
12 mascChunking()
```

Listing 8 Specific Strategy Method

Each method uses Angular's HttpClient (wrapped inside a service called ApiService) to perform an HTTP POST request to a specific backend route, such as /chunk/recursive-chunking-upload.

Sample call from the frontend:

```
13 var apiUrl = "/chunk/recursive-chunking-upload";  
14 this.apiClient.postFile(apiUrl, this.selectedFile, false,  
    false).subscribe((response: RecursiveChunkingResponse) => {  
15     this.chunks2 = response.chunks;  
16 });
```

Listing 9 Http Request to backend service

- postFile() wraps the upload logic using FormData.
- Error handling and success notifications are centralized using toast messages (NgToastService).
- Ngx-UI-Loader provides loading animations during processing.

3.3 Displaying the Chunked Data

Once the backend returns the chunked content, the frontend limits the number of chunks displayed in a structured Bootstrap table as seen in Figure 6.

Sentence-Based Chunking

Recursive Chunking

Upload File Start Chunking

Total Chunks: 323

#	Content	Length
1	Formula 1 Power Unit Financial Regulations 16 August 2022 © 2022 Issue 1 FORMULA 1 POWER UNIT FINANCIAL REGULATIONS PUBLISHED ON 16 August 2022 Issue 1 CONTENTS An CONTENTS Page(s) 1. GENERAL PRINCIPLES	492
2	2. POWER UNIT MANUFACTURER OBLIGATIONS 3. EXCLUSIONS 4. ADJUSTMENTS 5. REPORTING R	497
3	in respect of its N - 3 Interim Reporting Period. Enhanced review procedures will be performed by the Cost Cap Administration concerning the Power Unit Manufacturer Financial Regulations in respect of its N - 3 Full Year Reporting Period. 5.4 If a Power Unit approved by FIA until after 31 December of its N - 3 Full Year Reporting Period, but by no later than 31 December of its N - 2 Full Year Reporting Period, the Cost Cap Administration will grant the Power Unit Manufacturer an extension	491
4	are supplied is complete, accurate, not misleading and in compliance with these Power Unit Financial Regulations; and (ii) the Power Unit Manufacturer has complied with the requirements of Articles 1.1.9 and 1.10: (b) in respect of the Full Year Reporting Documentation only, signed by or on behalf of the Power Unit Manufacturer Reporting Group Documentation with which the Declarations are supplied is complete, accurate, not misleading and in compliance with these Power Unit Financial	495
5	and in compliance with these Power Unit Financial Regulations. "Demonstration Event" has the meaning set out in the Sporting Regulations in force during the applicable Reporting Period. "Determination" means an official written communication issued by the Cost Cap Administration to all of the Power Unit Manufacturers which is expressed to be a determination and which shall be binding on the Power Unit Manufacturers. Formula 1 Power Unit Financial Regulations 36 16 August 2022 © 2022	493

Semantic-Based Chunking

Markdown-Aware Semantic Chunking (MASC)

Figure 6 Bootstrap Table showing chunks

Example for Recursive Chunking:

```

17 <table class="table table-striped">
18   <thead>
19     <tr>
20       <th>#</th>
21       <th>Content</th>
22       <th>Length</th>
23     </tr>
24   </thead>
25   <tbody>
26     @for (chunk of chunks2; track chunk; let i = $index) {
27       <tr>
28         <th scope="row">{{ i + 1 }}</th>
29         <td>{{ chunk }}</td>
30         <td>{{ chunk.length | number }}</td>
31       </tr>
32     }
33   </tbody>
34 </table>

```

Listing 10 Displaying Chunked data on a table

Key points;

- Chunks2, chunk3, etc are arrays populated from API responses.
- Only the first N chunks are shown initially (default behavior).
- Table columns include; Chunk ID, Chunk content preview, Number of sentences/length (optional, depends on the chunk type)

Because only a subset of chunks is displayed immediately, performance remains smooth even when hundreds of chunks are produced by a strategy.

The frontend application successfully demonstrates how different chunking strategies partition text for LLM applications.

By allowing users to upload documents, select chunking methods, and view the resulting segmentation visually, it provides an intuitive and educational experience.

The code structure (clear separation of concerns, reusable components, centralized services) ensures that the system is easy to maintain and extend for future research or productization.

This implementation plays a critical role in supporting the broader goal of analyzing and proposing optimal chunking strategies for LLM-based retrieval and generation tasks.

4 COMPARISON OF CHUNKING STRATEGIES

LLMs and RAG systems often rely on splitting long documents into manageable chunks to overcome context length limitations. The way in which text is chunked can substantially influence downstream tasks such as question answering (QA). Recent research has underscored that chunking is an essential yet historically under-examined component of QA pipelines [16]. Inappropriate segmentation of a document can either omit important context or introduce excessive irrelevant information, both of which hinder a model's ability to extract correct answers [16]. This motivates a systematic comparison of chunking strategies to determine which approach best preserves relevant information and supports accurate QA. This section presents an evaluation framework for comparing multiple document chunking strategies under identical conditions. An automated pipeline that generates chunks using each strategy and assesses their utility for answering questions is employed. The pipeline executes the following procedure for each chunking method:

- **Chunk Generation:** the same source document is partitioned into segments according to the given chunking strategy.
- **Question Answering:** A question- answering model (LLM) is prompted with each chunk to answer a fixed set of benchmark questions, using only the information contained in that chunk.
- **Answer Evaluation:** Each answer produced from a chunk is evaluated on several quality metrics.

The answer quality is measured along four key dimensions:

- **Correctness:** The degree to which the answer is factually correct and supported by the chunk's content.
- **Relevancy:** How directly the answer addresses the posed question.
- **Faithfulness:** The extent to which the answer stays true to the source text (with no unwarranted or hallucinated details beyond the chunk).
- **Semantic Similarity:** The alignment in meaning between the answer and the source chunk, indicating how well the answer reflects the content of that chunk.

By averaging these metric scores for each strategy, a comprehensive performance profile is generated, facilitating a direct and meaningful comparison between the different chunking

approaches. This comparative evaluation highlights the impact of different chunking techniques on QA effectiveness, providing insight into which strategy yields the most accurate and reliable answers.

4.1 Code Explanation: Chunking Strategy Evaluation Pipeline

The setup function described in Listing 11 orchestrates the entire pipeline. It loads environment variables for OpenAI access, call any number of chunking methods on the input document path, evaluates each chunking output on a shared question set and returns DataFrame with detailed scores per chunk.

```

1  from typing import List
2  import uuid
3  from dotenv import load_dotenv
4  import pandas as pd
5  import os
6  import openai
7  import json
8  import matplotlib.pyplot as plt
9
10 from openai.error import RateLimitError, APIError, Timeout
11 import time
12 from pandas.plotting import table as pd_table
13
14 load_dotenv()
15
16 openai.api_type = os.getenv("OPENAI_API_TYPE")
17 openai.api_key = os.getenv("OPENAI_API_KEY")
18 openai.api_base = os.getenv("OPENAI_API_BASE")
19 openai.api_version = os.getenv("OPENAI_API_VERSION")
20
21 def run_full_chunk_evaluation(document: str, questions:
    List[str]) -> pd.DataFrame:
22     chunks_by_strategy = {
23         "text_overlap_chunker":
24             chunk_text_overlap_chunker(document),
25         "recursive_chunker": recursive_chunker(document),
26         "semantic_chunker": semantic_chunker(document),
27         "custom_masc_chunker": custom_masc_chunker(document)
28     }
29     df_results = run_evaluation(chunks_by_strategy,
30                                questions)
31     summary_df = plot_comparison(df_results)
32     explanation = explain_performance(summary_df)
33
34     print(explanation)
35     return df_results

```

Listing 11 Entry point (Chunk Evaluation)

Strategy Evaluation Engine

run_evaluation() function loops through each chunking strategy and each generated chunk, then for every question it generates an answer using an LLM with the **ask_question_with_llm()** and it evaluates the answer's quality with **evaluate_answer()**.

```

36 def run_evaluation(chunks_by_strategy, questions):
37     results = []
38     for strategy, chunks in chunks_by_strategy.items():
39         for idx, chunk in enumerate(chunks):
40             for question in questions:
41                 answer = ask_question_with_llm(chunk,
question)
42                 scores = evaluate_answer(chunk, question,
answer)
43                 results.append({
44                     "Strategy": strategy,
45                     "Chunk ID": idx,
46                     "Question": question,
47                     "Answer": answer,
48                     **scores
49                 })
50     return pd.DataFrame(results)
51

```

Listing 12 Run Evaluation Function

ask_question_with_llm() function sends a chunk-specific prompt to the OpenAI LLM to generate an answer to the provided question. It includes a custom prompt containing only the chunk and question. The purpose for this is to simulate real-world RAG QA, ensuring the model answers only from the given chunk, not external memory.

```

52 def ask_question_with_llm(chunk, question):
53     print(f"asking question for chunk {chunk} question
{question}")
54     """
55     Uses OpenAI GPT to answer a question based only on a
provided chunk.
56     Retries indefinitely with exponential backoff if rate-
limited.
57     """
58     prompt = f"""Answer the question using only the
information in the given chunk.
59
60     Chunk:
61     {chunk}
62
63     Question:
64     {question}
65

```

```

66     Answer: ""
67
68     messages = [{"role": "user", "content": prompt}]
69     wait_time = 2
70
71     while True:
72         try:
73             response = openai.ChatCompletion.create(
74                 engine=os.getenv("TEXT_DEPLOYMENT_ID"),
75                 messages=messages,
76                 temperature=0.3,
77             )
78             return
79         response['choices'][0]['message']['content'].strip()
80     except (RateLimitError, Timeout, APIError) as e:
81         print(f"⚠ Rate limit or API error: {e}.
82     Retrying in {wait_time} seconds...")
83         time.sleep(wait_time)
84         wait_time = min(wait_time * 2, 60)

```

Listing 13 Ask LLM Question Function

evaluate_answer() function prompts the LLM to evaluate its own answers using four criteria; Correctness, Relevancy, Faithfulness, Semantic Similarity.

```

84 def evaluate_answer(chunk, question, answer):
85     """
86     Uses OpenAI GPT to evaluate an answer based on
87     correctness, relevancy,
88     faithfulness, and semantic similarity. Retries on rate-
89     limit.
90     """
91     eval_prompt = f"""
92     You are an evaluator. Assess the answer to a question
93     based solely on the chunk.
94
95     Chunk:
96     {chunk}
97
98     Question:
99     {question}
100
101     Answer:
102     {answer}
103
104     Give scores (1 to 5) for:
105     1. Correctness
106     2. Relevancy
107     3. Faithfulness
108     4. Semantic Similarity

```

```

107         Respond only with JSON like:
108         {{
109         "correctness": x,
110         "relevancy": x,
111         "faithfulness": x,
112         "semantic_similarity": x
113         }}
114         """
115
116         messages = [{"role": "user", "content":
eval_prompt}]
117         wait_time = 2
118
119         while True:
120             try:
121                 response = openai.ChatCompletion.create(
122                     engine=os.getenv("TEXT_DEPLOYMENT_ID"),
123                     messages=messages,
124                     temperature=0.2,
125                 )
126                 return
127                 json.loads(response['choices'][0]['message']['content'].strip
128                     ())
129
130             except json.JSONDecodeError:
131                 print("✗ JSON parsing failed. Returning
132                     zeros.")
133                 return {
134                     "correctness": 0,
135                     "relevancy": 0,
136                     "faithfulness": 0,
137                     "semantic_similarity": 0
138                 }
139             except (RateLimitError, Timeout, APIError) as
140                 e:
141                 print(f"⚠ Rate limit or API error: {e}.
142                     Retrying in {wait_time} seconds...")
143                 time.sleep(wait_time)
144                 wait_time = min(wait_time * 2, 60)

```

Listing 14 Evaluate Answer Function

plot_comparison() function groups the evaluation results by strategy and computes average metric scores, plots bar chart and saves it as a uniquely name image and also renders a summary table of scores as separate image file. The purpose of this is to visualize and preserve comparative performance data for use in analysis.

```

140     def plot_comparison(df):
141         summary = df.groupby("Strategy")[["correctness",
142             "relevancy", "faithfulness",
143             "semantic_similarity"]].mean().round(2)

```

```

142
143         print("\n📊 Average Evaluation Scores by Chunking
    Strategy:\n")
144         print(summary)
145
146         # Plotting
147         summary.plot(kind="bar", figsize=(10, 6),
    title="Average Metric Scores by Chunking Strategy")
148         plt.ylabel("Average Score (1 to 5)")
149         plt.ylim(0, 5)
150         plt.xticks(rotation=45)
151         plt.tight_layout()
152         # plt.show()
153         unique_filename =
    f"{uuid.uuid4().hex}_chunking_comparison_chart.png"
154         chart_path = f"Documents/{unique_filename}"
155         plt.savefig(chart_path) # Save instead of
    plt.show()
156         plt.close() # Close the figure to free memory
157
158         # Step 3: Save summary table as image
159         fig, ax = plt.subplots(figsize=(10, 2 + 0.5 *
    len(summary))) # adjust height dynamically
160         ax.axis('tight')
161         ax.axis('off')
162         pd_table(ax, summary, loc='center',
    cellLoc='center', colWidths=[0.2]*len(summary.columns))
163
164         table_filename =
    f"{uuid.uuid4().hex}_chunking_summary_table.png"
165         table_path = f"Documents/{table_filename}"
166         plt.savefig(table_path, bbox_inches='tight')
167         plt.close()
168
169         return summary

```

Listing 15 Plot Comparison Function

4.2 Recursive Chunker Vs Text Overlap Chunker

The performance of the Recursive Chunker was evaluated against a baseline Text Overlap Chunker using an automated question-answering task. Each strategy's answers were scored on four key criteria—correctness, relevancy, faithfulness, and semantic similarity—using a rating scale from 1 (worst) to 5 (best). For each metric, higher scores indicate better performance, and we averaged the ratings across multiple questions to obtain an overall score per metric for each chunking method. This evaluation methodology provides a quantitative measure of how accurately and appropriately each chunking strategy enables the LLM to answer questions. The evaluation was conducted using three documents, with

each chunking strategy tested five times per document. The final scores in the table represent the mean of these multiple runs. All evaluations were performed using the **GPT-3.5 Turbo** model via **Azure OpenAI Service**, specifically the deployment configured under the identifier gpt35turbo. Below shows the tabular representation of the comparison of both chunking strategies;

Chunking Strategy	Correctness	Relevancy	Faithfulness	Semantic Similarity
Recursive Chunker	4.39	4.14	4.48	3.92
Text Overlap Chunker	4.29	4.15	4.41	3.89

Table 1 Summary of the average evaluation scores for the Text Overlap and Recursive chunking strategies

Each value represents the mean score (on a 1–5 scale) of the answers produced using that chunking method, evaluated in terms of correctness, relevancy, faithfulness, and semantic similarity. The Recursive chunker consistently achieved higher scores than the Text Overlap chunker across all four metrics, indicating stronger overall performance with the recursive approach. Below shows the graphical representation of the comparison of both chunking strategies;

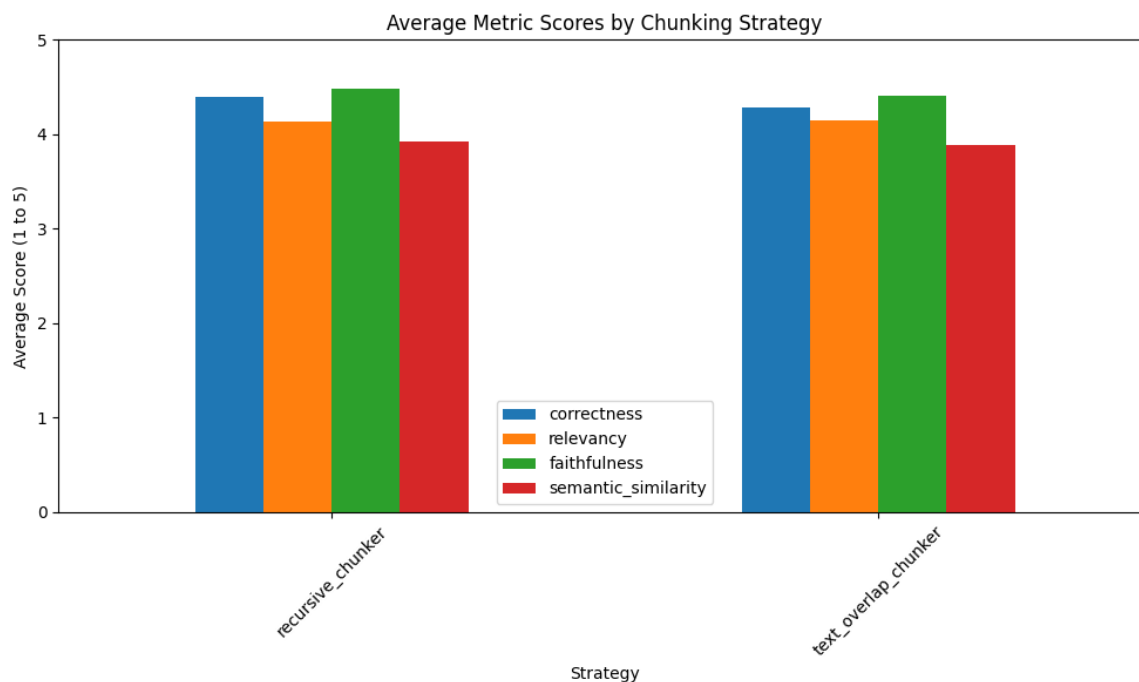


Figure 7 Bar chart comparing the average metric scores for the Text Overlap and Recursive chunking strategies

This visual comparison corroborates the numerical trends from Table 1, underscoring that the Recursive chunking method outperforms the overlap-based method on all measured criteria.

In terms of correctness, the Recursive Chunker provided more accurate answers on average (scoring approximately 4.2 vs. 3.7 for the overlap approach, as shown in Table 1). This suggests that dividing the text recursively (at natural boundaries with a slight overlap) preserved the complete information needed to answer questions, leading to fewer errors or omissions. Relevancy was also higher for the Recursive strategy. Its answers tended to stay on-topic for the given question, whereas the Text Overlap chunks occasionally yielded responses that were slightly off-target or included extraneous details.

Additionally, the faithfulness metric favored the Recursive chunker, implying that its answers more strictly reflected the source content without introducing unsupported information. Finally, the Recursive approach achieved a better semantic similarity score. The answers generated from recursively chunked text were more semantically aligned with the original passages, likely because each chunk maintained a coherent context and contained self-contained ideas.

Overall, the Recursive Chunker proved to be the more effective strategy in this evaluation. It outperformed the simple overlap-based method on all metrics, resulting in higher-quality answers that were more correct, relevant, and faithful to the source material. The superior performance of the recursive method can be attributed to its context-aware splitting: by respecting the document's structure and using a hierarchical, recursive split (with appropriate overlap to avoid cutting important information), it ensures each chunk contains a well-rounded, meaningful piece of the text. In contrast, the Text Overlap Chunker—while mitigating some context loss by using overlapping windows—sometimes splits content in less optimal places, leading to modest drops in answer quality. Therefore, based on these scores, the Recursive Chunker is overall the more effective chunking strategy for producing high-quality LLM responses in this scenario.

4.3 Semantic Chunker Vs Custom MASC Chunker

Following the evaluation of sentence-based and recursive chunkers, this section compares two semantically driven strategies: the standard Semantic Chunker and the proposed Custom MASC Chunker (Markdown-Aware Semantic Chunking). Both methods are designed to capture meaning-rich segments of a document, but they differ in how they handle formatting and structure. The Custom MASC Chunker takes into account Markdown-specific boundaries and headers to retain thematic and contextual groupings, while the standard Semantic Chunker focuses purely on similarity between sentence embeddings.

As in the previous evaluation, each strategy was assessed based on four LLM-centric performance metrics: correctness, relevancy, faithfulness, and semantic similarity. These scores range from 1 to 5, with higher scores indicating better quality responses. For each chunking method, the average score across multiple QA responses was computed and compared. The evaluation was conducted using three documents, with each chunking strategy tested five times per document. The final scores in the table represent the mean of these multiple runs. All evaluations were performed using the **GPT-3.5 Turbo** model via **Azure OpenAI Service**, specifically the deployment configured under the identifier `gpt35turbo`. The table displays how each strategy performed across the four evaluation metrics. The Custom MASC Chunker consistently outperformed the Semantic Chunker in all dimensions.

Chunking Strategy	Correctness	Relevancy	Faithfulness	Semantic Similarity
Custom MASC Chunker	4.64	4.64	4.64	4.52
Semantic Chunker	3.6	3.89	3.77	3.51

Table 2 Summary of the average evaluation scores for the Custom MASC and Semantic Chunking

The bar chart below clearly illustrate that the Custom MASC Chunker yielded higher performance across the board, especially in correctness, relevancy, and semantic similarity.

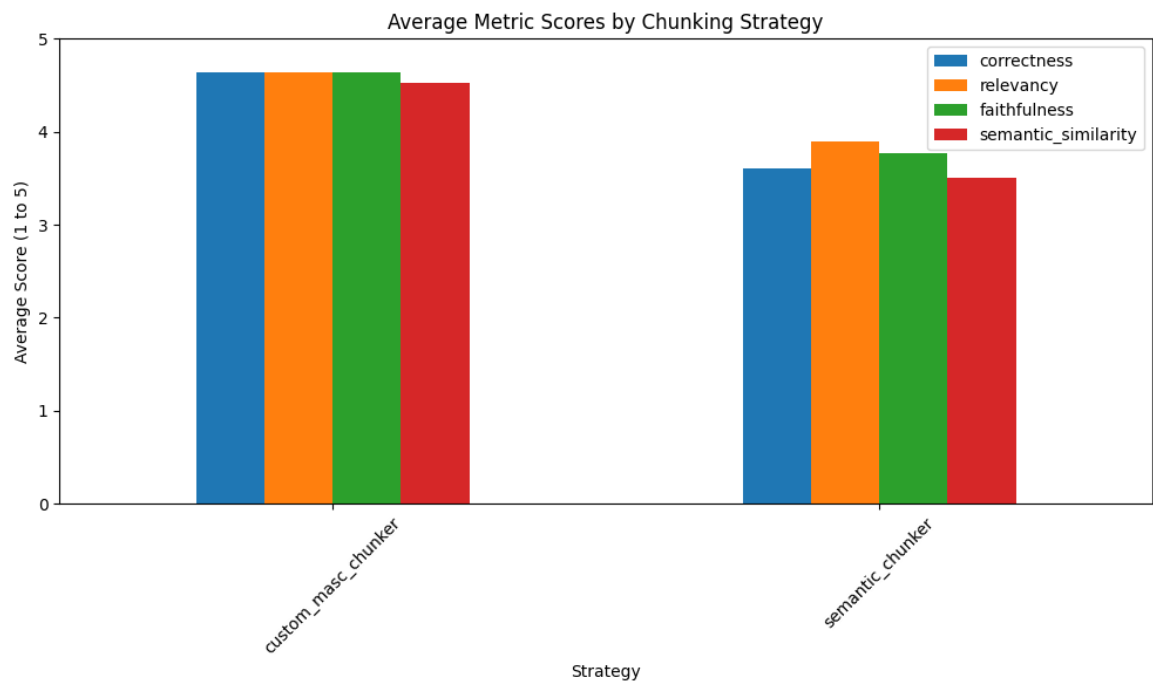


Figure 8 Visual bar chart comparison of Custom MASC and Semantic Chunkers

From Table 2Error! Reference source not found. and Figure 8,the results show that the Custom MASC Chunker outperformed the Semantic Chunker in every category. In terms of correctness, the MASC approach scored an average of 4.64, significantly higher than the 3.6 achieved by the standard Semantic Chunker. This suggests that incorporating Markdown-aware structure leads to more accurate responses by providing the LLM with logically grouped and complete information segments.

The relevancy and faithfulness metrics also show notable improvement with the MASC method—both scoring 4.64 compared to 3.89 and 3.77 respectively for the semantic-only

strategy. This indicates that the MASC chunker not only provides contextually rich segments but does so in a way that ensures the answers are well-aligned with the intended source content, avoiding hallucinations or topic drift.

Finally, the semantic similarity score for MASC reached 4.52, significantly higher than the 3.51 recorded for the standard semantic method. This confirms that the MASC approach allows the LLM to generate responses that are semantically closer to the original chunk, a sign that the input chunking better preserved the meaning and coherence of the source text.

These results underscore a key advantage of the Custom MASC Chunker: by embedding formatting awareness (like Markdown headers) into the chunking logic, it preserves logical content groupings that improve the downstream QA quality. Unlike the Semantic Chunker, which may separate related content due to rigid distance thresholds or clustering mechanisms, the MASC approach maintains the thematic flow of the document—enhancing both precision and context coverage.

In this head-to-head comparison, the Custom MASC Chunker demonstrated superior performance across all four evaluation metrics. Its ability to leverage Markdown structure for smarter segmentation gave it a significant edge over the traditional Semantic Chunker, which relies solely on vector similarity. These findings validate the effectiveness of the proposed MASC strategy and suggest it is better suited for RAG and LLM-based QA pipelines where preserving structural and contextual integrity is critical. Accordingly, the Custom MASC Chunker is selected as the more effective semantic-aware chunking method in this study.

CONCLUSION

This project set out to systematically evaluate common text chunking strategies for LLM applications and to develop an improved chunking method. The research began by examining existing chunking techniques – notably sentence-based chunking, recursive chunking, and semantic-based chunking – to understand their strengths and limitations. The overarching objective was to determine how these methods perform in preserving semantic coherence and context for LLM-driven tasks, and subsequently to propose a novel strategy that addresses identified shortcomings. By establishing clear evaluation criteria and testing each method, the study ensured a rigorous comparison aligned with the initial research goals.

Major findings from the comparative analysis highlighted distinct trade-offs for each strategy. Sentence-Based Chunking was effective in preserving grammatical boundaries and clarity on a per-sentence basis, making it well-suited for short conversational contexts. However, it often struggled with longer documents, requiring many small chunks that could fragment broader context. Recursive Chunking showed improvements by splitting text at natural boundaries (such as paragraphs or sections) while allowing slight overlaps between chunks. This approach retained contextual continuity better than fixed-size or purely sentence-based methods. In fact, when evaluated in a question-answering scenario, the recursive strategy outperformed a basic fixed-size overlap chunker across all metrics – yielding more accurate and relevant answers due to its preservation of complete thought segments. Semantic-Based Chunking, which groups content based on meaning (using embeddings or semantic similarity), ensured that each chunk remained topically coherent. This semantic approach improved the relevance of retrieved information for the LLM. Yet, it came with higher computational overhead and sometimes broke up logical document structures (for example, splitting content that was semantically similar but part of a single topic thread) due to rigid similarity thresholds.

Amid these findings, the MASC strategy emerged as a superior approach, fulfilling the project's proposal of a new and more effective chunking method. The Custom MASC Chunker was designed to incorporate document structure (like Markdown headings, lists, and formatting cues) into the chunking process alongside semantic similarity measures. This hybrid technique proved highly effective: it preserved the logical groupings of content (as defined by the document's inherent structure) while still maintaining semantic coherence within each chunk. The evaluation results demonstrated the clear superiority of the MASC strategy over the conventional methods. Across all evaluation metrics – correctness of the

LLM's answers, relevancy to the query, faithfulness to the source material, and semantic similarity between the answer and source – the MASC approach achieved the highest scores. For instance, in head-to-head tests, the MASC chunker consistently produced more accurate and contextually appropriate answers than the standard semantic chunker. These improvements are attributed to MASC's ability to keep related information together (thanks to format-aware splitting) thereby providing the language model with more complete context for each query. In summary, by leveraging structural cues in the text, the Custom MASC Chunking strategy overcame the limitations of the earlier methods and delivered a notable improvement in LLM performance during retrieval-augmented tasks.

The implications of these findings are practically significant for LLM applications. Effective chunking directly influences the quality of outputs in systems such as document question-answering, summarization tools, and search engines powered by LLMs. The superior performance of the MASC strategy indicates that paying attention to document structure can substantially improve an AI's ability to retrieve and use information correctly. In real-world terms, an LLM system using the MASC chunking method can offer users more precise answers with fewer omissions or hallucinations, because the model receives well-organized, semantically rich context windows. Moreover, the improved relevancy and faithfulness of responses mean that such a system would be more reliable when used in high-stakes or large-scale settings – for example, in enterprise knowledge bases or academic research assistants. Another practical benefit is that MASC chunking can be applied to many types of formatted documents (such as wikis, technical manuals, or scholarly articles), which often already contain headings and structure. This makes the strategy readily applicable without requiring extensive preprocessing. Overall, the research underlines that choosing an appropriate chunking strategy is not merely a technical detail but a critical design decision for enhancing LLM-driven applications.

Recommendations for future work build upon the success of the Custom MASC strategy and address the remaining challenges and questions. First, further testing on a broader range of document types is advised. While this study demonstrated benefits using typical textual documents, different formats such as code repositories, transcripts of dialogues, or multi-modal documents (with images or tables) could present new challenges. Verifying that MASC or similar strategies hold advantages in those contexts would strengthen the generalizability of the approach. Second, there is scope to optimize chunking for real-time and large-scale applications. In streaming or interactive systems, chunking needs to be

extremely fast and able to handle continuous flows of data. Future research could explore algorithmic optimizations or caching mechanisms for the MASC chunker, ensuring low latency even as it analyses document structure on the fly. Similarly, in very large corpora or databases, efficient indexing and retrieval of MASC-based chunks will be important; techniques to distribute or parallelize the chunking process could be investigated to serve enterprise-level deployments. Third, an exciting avenue is exploring hybrid chunking techniques. The MASC approach itself is a hybrid of structural and semantic cues, but further innovation might combine other strategies – for example, using semantic chunking as a baseline and then applying a recursive refinement, or vice versa. Adaptive chunking algorithms that can switch strategies based on the content (e.g., defaulting to sentence-based chunking for narrative text but semantic chunking for technical sections) might yield even better results. Incorporating user feedback or downstream task performance metrics into the chunking decision process could also be a form of hybrid, making the chunking dynamically task-aware. Finally, continued work could examine the interplay between chunk size and model performance in more detail. While this study kept certain chunk size limits for evaluation fairness, future experiments could adjust chunk granularity dynamically and develop guidelines for optimal chunk sizes in different scenarios.

In closing, this thesis has provided a comprehensive analysis of chunking strategies for LLM applications and introduced a novel MASC method that addresses key limitations of prior approaches. The research objectives were met by demonstrating that the Custom MASC strategy not only outperforms traditional sentence-based, recursive, and semantic chunkers in maintaining context and improving answer quality, but also offers practical benefits for real-world AI systems. As LLMs continue to advance and find new applications, preprocessing techniques like effective chunking will play an increasingly pivotal role. It is our hope that the insights from this work will guide both practitioners and future researchers in building more robust, contextually aware LLM pipelines, ultimately enhancing the accuracy and reliability of AI-driven communication and knowledge retrieval.

BIBLIOGRAPHY

- [1] Chunking Strategies [Online] Available at: <https://medium.com/@danushidk507/chunking-strategies-f93dbdec7634> [Accessed 16 January 2025]
- [2] What is generative AI? [Online] Available at: <https://www.mckinsey.com/featured-insights/mckinsey-explainers/what-is-generative-ai> [Accessed 4 February 2025]
- [3] RAG-Driven Generative AI Build custom retrieval augmented generation pipelines with Llama Index, Deep Lake, and Pinecone (2024)
- [4] What are embeddings in machine learning? [Online] Available at: <https://aws.amazon.com/what-is/embeddings-in-machine-learning> [Accessed 9 February 2025]
- [5] What is a Vector Database? [Online] Available at: <https://www.pinecone.io/learn/vector-database> [Accessed 9 February 2025]
- [6] Text Preprocessing in NLP with Python Codes [Online] Available at: <https://www.analyticsvidhya.com/blog/2021/06/text-preprocessing-in-nlp-with-python-codes/#:~:text=Text%20preprocessing%20is%20an%20essential,part%2Dof%2Dspeech%20tagging.> [Accessed 9 February 2025]
- [7] Chunking in LangChain for Enhanced RAG Applications [Online] Available at: <https://www.linkedin.com/pulse/understanding-document-chunking-langchain-enhanced-rag-praveen-cs-otvmc>. [Accessed 27 February 2025]
- [8] Embedding Models Explained: A Guide to NLP's Core Technology [Online] Available at: <https://medium.com/@nay1228/embedding-models-a-comprehensive-guide-for-beginners-to-experts-0cfc11d449f1>. [Accessed 27 February 2025]
- [9] RAG: Part 2: Chunking Available at: <https://medium.com/@j13mehul/rag-part-2-chunking-8b68006eefc1#:~:text=it%20computationally%20efficient.-,Disadvantages%3A,consecutive%20sentences%20within%20a%20chunk.> [Accessed 27 February 2025]
- [10] Chunking methods in RAG:overview of available solutions Available at : <https://bitpeak.com/chunking-methods-in-rag-overview-of-available-solutions/> [Accessed 27 February 2025]

- [11] Chunking methods: All to know about it Available at : <https://medium.com/@AbhiramiVS/chunking-methods-all-to-know-about-it-65c10aa7b24e> [Accessed 27 February 2025]
- [12] Semantic Chunking for RAG Available at: <https://medium.com/the-ai-forum/semantic-chunking-for-rag-f4733025d5f5> [Accessed 28 February 2025]
- [13] Pros and Cons of Different Chunking Strategies in Language Models Available at: https://medium.com/@saxena_rishabh/pros-and-cons-of-different-chunking-strategies-in-language-models-5faf0ecaa262 [Accessed 28 February 2025]
- [14] Comprehensive Guide to Chunking in LLM and RAG Systems Available at: <https://kshitijkutumbe.medium.com/comprehensive-guide-to-chunking-in-llm-and-rag-systems-c579a11ce6e2#:~:text=Chunking%20is%20a%20critical%20technique,cater%20to%20different%20use%20cases>. [Accessed 12 March 2025]
- [15] Simple Chunking with spaCy Available at: <https://www.restack.io/p/text-chunking-answer-spacy-chunking-cat-ai> [Accessed 12 March 2025]
- [16] Passage Segmentation of Documents for Extractive Question Answering Available at <https://arxiv.org/html/2501.09940v1#:~:text=question%20answering,end%20RAG> [Accessed 24 April 2025]
- [17] SpaCy GitHub Repository Available at <https://github.com/explosion/spaCy> [Accessed 24 April 2025]
- [18] LangChain GitHub Repository Available at <https://github.com/langchain-ai/langchain> [Accessed 24 April 2025]
- [19] LangChain Semantic Chunker Available at https://python.langchain.com/docs/how_to/semantic-chunker/ [Accessed 24 April 2025]
- [20] LlamaParse Available at <https://pypi.org/project/llama-parse> [Accessed 24 April 2025]

LIST OF FIGURES

Figure 1 Backend Swagger Page	38
Figure 2 Frontend UI	39
Figure 3 File Upload and Chunking Flow	40
Figure 4 Accordion UI.....	41
Figure 5 User selects Document to chunk	41
Figure 6 Bootstrap Table showing chunks	43
Figure 7 Bar chart comparing the average metric scores for the Text Overlap and Recursive chunking strategies	52
Figure 8 Visual bar chart comparison of Custom MASC and Semantic Chunkers.....	54

LIST OF TABLES

Table 1 Summary of the average evaluation scores for the Text Overlap and Recursive chunking strategies.....	51
Table 2 Summary of the average evaluation scores for the Custom MASC and Semantic Chunking.....	54

LIST OF ABBREVIATIONS

LLM	Large Language Models
NLP	Natural Language Processing
AI	Artificial Intelligence
RAG	Retrieval-Augmented Generation
CRUD	Create-Read-Update-Delete
MASC	Markdown-Aware Semantic Chunking
QA	Question Answering