# 大语言模型数据准备中的实用分块技术、开源框架及应用

## I. 引言：大语言模型数据准备中分块技术的重要性

### A. 分块的定义及其在 LLM 应用中的核心作用

分块（Chunking）是一种在大型语言模型（LLM）应用中将冗长或复杂的文本分解为更小、更易于管理的部分（称为“块”或“chunks”）的方法 1。此过程至关重要，因为它能帮助 LLM 更有效地处理信息，而不会遗漏重要的细节或上下文 1。鉴于许多 LLM 存在处理文本长度的限制（通常以“token”或字符数衡量），分块确保了输入数据能够适应这些限制，同时仍然能够产生准确且相关的结果 1。

在 LLM 应用中，分块的核心作用体现在多个方面。首先，它通过将大规模文本数据分解为逻辑上连贯的小单元，优化了数据处理流程并提升模型性能 2。其次，分块有助于提高模型的准确性、维持上下文连贯性，并增强问答、摘要和搜索等任务的整体表现 1。若无分块，模型可能会忽略关键细节或返回不完整的响应，尤其在处理如研究论文、法律文件或客户支持日志等长文档时，模型可能因输入超限而无法完整处理或丢失重要信息 1。因此，精心设计的分块策略能够确保 LLM 应用运行得更快、更智能、更可靠 1。

### B. 有效分块策略对模型性能的影响

有效的文本分块策略对大型语言模型（LLM）的性能具有显著的正面影响。设计良好的分块策略能够在每个文本片段中保留原始文本的意义、上下文和相关性，从而引导模型产出更高质量的结果 1。具体而言，其影响体现在以下几个关键方面：

1. **提升准确性和相关性**：通过将长文本分解为更小、更集中的片段，LLM 可以更精确地处理每个部分，确保对各分段内容的彻底理解 3。这使得模型能够生成更准确、与上下文更相关的响应，尤其在处理包含大量信息的复杂文档时，分块有助于模型关注每个文本块的具体内容，从而生成更精确的答案 1。
2. **维持上下文连贯性**：尽管分块将文本拆分，但优秀的分块策略（如语义分块或使用重叠块）致力于在块与块之间保持上下文的连续性 1。这对于理解跨越多个文本片段的复杂叙述或论点至关重要，有助于模型生成连贯的回答并提取关键信息 2。
3. **优化检索质量与效率**：在检索增强生成（RAG）等应用中，分块是确保检索内容相关性的关键因素。它通过将大型文件分割成易于管理的小块，提升了上下文的准确性，使得检索系统能够更高效地定位和提取与用户查询最相关的文本片段 4。这不仅提高了检索内容的质量，还通过减少需要处理的文本量来降低延迟，并优化存储成本 7。
4. **有效管理Token限制**：LLM通常具有输入Token数量的上限。分块确保了输入文本符合这些限制，避免了因超长输入导致的信息截断或处理失败 1。通过明智地切分文本，可以在遵循Token限制的同时最大程度地保持信息的完整性 9。
5. **增强特定任务的性能**：分块策略应与LLM应用的目标相一致。例如，问答系统受益于包含丰富上下文的较小文本块，以便快速检索和解释；文档摘要则更适合能够捕捉更多内容流的较大文本块；而语义搜索需要能够保留主题边界和富含关键词片段的文本块 1。

综上所述，一个经过深思熟虑的分块策略是LLM应用成功的基石，它直接影响着模型理解、处理信息的能力以及最终输出的质量和效率 1。

### C. 本报告的目的与结构

本报告旨在全面总结和阐述在大型语言模型（LLM）数据准备阶段所需的实用分块技术。随着LLM在自然语言处理领域的广泛应用，如何有效地预处理和组织输入数据，特别是长文本数据，已成为提升模型性能和应用效果的关键环节。分块作为核心的数据准备步骤，其策略选择与实施对模型的上下文理解、信息检索的准确性以及生成结果的质量具有直接且深远的影响。

本报告将系统性地介绍当前主流的文本分块方法，涵盖其工作原理、优缺点、适用场景，并结合具体的开源框架（如Langchain, LlamaIndex, NLTK, spaCy等）提供实际应用的代码示例和说明。具体结构安排如下：

* **第二部分：基础分块技术** - 详细介绍固定大小分块和递归分块这两种基本且常用的方法。
* **第三部分：内容感知分块技术** - 探讨更侧重文本内在逻辑的句子切分和语义分块技术。
* **第四部分：针对结构化和半结构化数据的专用分块技术** - 介绍为Markdown、HTML、代码和LaTeX等特定格式设计的专用分块器。
* **第五部分：高级分块策略和新兴概念** - 讨论如代理分块、块重叠的重要性、块大小优化，以及后期分块和上下文检索等前沿理念。
* **第六部分：实践指南：选择和实施正确的分块策略** - 提供选择分块策略的实用建议，并辅以一个比较概览表。
* **第七部分：结论与未来展望** - 总结关键技术，并展望分块技术未来的发展趋势和新的评估方法。

通过本报告，期望为LLM开发者和研究人员在数据准备阶段选择和应用分块技术提供清晰的指引和有价值的参考。

## II. 基础分块技术

基础分块技术为处理大型文本提供了初步且直接的方法，主要包括固定大小分块和递归分块。这些技术因其相对简单和易于实现而被广泛采用。

### A. 固定大小分块 (Fixed-Size Chunking)

#### 1. 工作原理

固定大小分块是最直接的文本分割方法之一 10。其核心原理是将文本按照预先设定的固定长度进行切割，这个长度可以基于字符数、词数或token数量来定义 2。例如，可以将一个文档分割成每块包含512个token的片段 9。这种方法不考虑文本的语义结构或自然边界，仅仅确保每个块的大小统一 3。为了在一定程度上保持块与块之间的上下文连续性，开发者有时会在相邻块之间设置重叠（overlap），即一个块的末尾部分会作为下一个块的起始部分重复出现 12。

#### 2. 优点

固定大小分块的主要优点在于其实现简单且计算效率高 7。由于块大小统一，管理和处理起来相对容易 7。这种方法对于那些文本结构一致或缺乏明显自然断点的应用场景（如产品描述或新闻文章）是合适的 1。

#### 3. 缺点

尽管简单高效，固定大小分块的主要缺点是它往往会忽略文本的内在语义结构 3。因为它不考虑句子、段落或主题的边界，所以很容易在不恰当的位置切断文本，比如将一个完整的句子或一个连贯的思绪分割到不同的块中 1。这可能导致上下文信息的丢失或语义的割裂，从而影响LLM对文本的理解和后续任务（如问答或摘要）的准确性 1。

#### 4. 常见用例

固定大小分块适用于处理那些结构相对简单、上下文连续性要求不高的文本数据 13。当数据缺乏自然的逻辑断点，或者需要对大量数据进行快速、统一处理时，这种方法可以发挥其优势 1。例如，在一些需要对大量结构化或半结构化数据（如日志文件、某些类型的表格数据转换成的文本）进行初步分割以适应模型输入限制的场景中，固定大小分块是一个可行的选择 2。

#### 5. 使用 Langchain 实现

Langchain框架提供了CharacterTextSplitter类来实现固定大小分块。此分割器可以基于字符数进行分割，并允许设置块大小（chunk\_size）和重叠量（chunk\_overlap）。

* **基于字符的固定大小分块 12:**  
  Python  
  from langchain\_text\_splitters import CharacterTextSplitter  
    
  long\_text = "This is a very long string of text that needs to be split into smaller chunks. Fixed-size chunking is a basic method. It will divide this text based on character count. We can also add some overlap between chunks to maintain context. For example, a chunk\_size of 50 and chunk\_overlap of 10."  
    
  # Initialize CharacterTextSplitter  
  text\_splitter = CharacterTextSplitter(  
   separator = " ", # 以空格作为基础分隔符，然后按chunk\_size组合  
   chunk\_size = 50,  
   chunk\_overlap = 10,  
   length\_function = len # 按字符长度计算块大小  
  )  
  docs = text\_splitter.create\_documents([long\_text])  
    
  for i, doc in enumerate(docs):  
   print(f"Chunk {i+1}: {doc.page\_content}")
* 基于Token的固定大小分块 1:  
  为了更精确地匹配LLM的token限制，可以使用与目标模型一致的tokenizer来定义块大小。Langchain的CharacterTextSplitter可以通过from\_tiktoken\_encoder（常用于OpenAI模型）或from\_huggingface\_tokenizer（用于自定义Hugging Face tokenizer）方法来实现基于token的分割。  
  Python  
  from langchain\_text\_splitters import CharacterTextSplitter  
  # 假设 document\_content 是要分割的文本  
  # document\_content = "Some very long text..."  
    
  # 使用tiktoken (适用于OpenAI模型)  
  # text\_splitter\_token = CharacterTextSplitter.from\_tiktoken\_encoder(  
  # encoding\_name="cl100k\_base", # OpenAI模型常用的编码器  
  # chunk\_size=100, # 每个块的目标token数量  
  # chunk\_overlap=20 # token重叠量  
  # )  
  # texts\_token\_based = text\_splitter\_token.split\_text(document\_content)  
    
  # 或者使用Hugging Face tokenizer [10]  
  # from transformers import AutoTokenizer  
  # tokenizer = AutoTokenizer.from\_pretrained("bert-base-uncased") # 示例tokenizer  
  # text\_splitter\_hf = CharacterTextSplitter.from\_huggingface\_tokenizer(  
  # tokenizer,  
  # chunk\_size=100,  
  # chunk\_overlap=20  
  # )  
  # texts\_hf\_based = text\_splitter\_hf.split\_text(document\_content)  
    
  (注意: 上述基于token的示例代码需要取消注释并提供document\_content才能运行。选择from\_tiktoken\_encoder或from\_huggingface\_tokenizer取决于所使用的LLM和tokenizer。)

### B. 递归分块：一种分层方法

#### 1. 工作原理

递归分块（Recursive Chunking）是一种更为精细的文本分割策略，它尝试通过一个预定义的、按优先级排列的分隔符列表（例如，常见的顺序是 ["\n\n", "\n", " ", ""]，分别代表段落、行、空格和空字符）来迭代地切分文本 3。其工作方式是：首先尝试使用列表中的第一个（通常是最大的语义单元分隔符，如双换行符代表段落）来分割文本。如果分割后的块仍然大于预设的 chunk\_size，那么这些过大的块将进一步使用列表中的下一个分隔符（如单个换行符代表句子或行）进行分割 16。这个过程会递归地进行下去，直到所有块都大致符合 chunk\_size 的要求，或者已尝试完所有分隔符（最终可能按字符分割）16。这种分层的方法旨在尽可能地保持较大语义单元（如段落，然后是句子，再然后是词语）的完整性 7。

#### 2. 优点

相比固定大小分块，递归分块在保持文本结构和意义方面表现更佳 7。它能够更灵活地适应文本内容的自然结构，因为它试图在自然的断点（如段落或句子结束处）进行分割 3。这通常能在维持语义连贯性和控制块大小之间取得较好的平衡，因此被认为是处理通用文本的推荐方法 14。

#### 3. 缺点

递归分块的一个主要缺点是，它不保证最终的块大小严格小于指定的最大值 7。因为算法会优先考虑使用分隔符来保持语义单元的完整性，所以如果一个不可再分的语义单元（例如一个长句子）本身就超过了 chunk\_size，它可能会被保留为一个略大的块。此外，分块的效果在很大程度上取决于所选择的分隔符列表及其顺序的合理性 7。配置一个有效的分隔符层级可能比固定大小分块略微复杂 14。

#### 4. 常见用例

递归分块非常适用于处理通用的非结构化文本，特别是当保持段落和句子的完整性对后续的LLM理解至关重要时 17。它也常用于处理篇幅较长、结构复杂或具有层级性的文档，例如技术手册、研究论文或法律合同等 2。

#### 5. 使用 Langchain 实现: RecursiveCharacterTextSplitter

Langchain 框架中的 RecursiveCharacterTextSplitter 是实现递归分块的推荐工具 17。它通过一个 separators 参数（一个字符串列表）来定义分割的层级顺序。

* **解释:** 该分割器会按 separators 列表中的顺序尝试分割文本。默认的 separators 列表是 ["\n\n", "\n", " ", ""]。
* **参数:** 主要参数包括 chunk\_size（目标块大小，通常按字符数计算），chunk\_overlap（块之间的重叠字符数），separators（分隔符列表），以及 length\_function（计算块长度的函数，默认为 len）10。
* **代码示例 10:**  
  Python  
  from langchain\_text\_splitters import RecursiveCharacterTextSplitter  
    
  # 包含段落和句子的示例文本  
  long\_text = "This is the first paragraph.\n\nThis is the second paragraph. It has multiple sentences. This is the third sentence of the second paragraph.\n\nThis is a very short third paragraph."  
    
  text\_splitter = RecursiveCharacterTextSplitter(  
   chunk\_size = 100, # 每个块的目标字符数  
   chunk\_overlap = 20,  
   length\_function = len,  
   separators=["\n\n", "\n", " ", ""] # 尝试按段落、行、空格顺序分割  
  )  
  docs = text\_splitter.create\_documents([long\_text])  
    
  for i, doc in enumerate(docs):  
   print(f"Chunk {i+1}: {doc.page\_content}")  
  正如 17 中所述：“这个文本分割器是通用文本的推荐选择。它由一个字符列表参数化。它会按顺序尝试在这些字符上进行分割，直到块足够小。默认列表是 [\"\\n\\n\", \"\\n\", \" \", \"\"]。这样做的效果是尽可能长时间地将所有段落（然后是句子，然后是单词）保持在一起，因为这些通常似乎是语义上最相关的文本片段。”

在基础分块技术中，固定大小分块因其简便快捷而被广泛应用，尤其是在对处理速度有较高要求且文本结构相对单一的场景中 2。然而，这种方法的代价是可能牺牲文本的语义完整性，因为它严格按照预设长度切分，容易割裂自然的语义单元，如句子或段落 3。递归分块则试图弥补这一缺陷，它通过一个预设的、具有优先级的字符列表（如段落分隔符优先于句子分隔符）来尝试分割文本，力求在达到目标块大小的同时，尽可能保持较大语义单元的完整性 7。这体现了在分块实践中一个根本性的权衡：计算简易性与语义保真度之间的平衡。应用开发者需要根据具体需求——是更看重处理效率，还是更强调意义的完整保留——来做出选择。对于许多通用文本处理任务，递归分块提供了一个在复杂性和语义保持之间的实用折中方案。

进一步审视递归分块中 chunk\_size 参数的作用，可以发现一个有趣的现象。尽管用户可以指定 chunk\_size，但实际分块结果并不严格保证所有块都小于此值 7。这是因为递归分块的核心机制是优先遵循 separators 列表进行分割。如果一个通过某个高级别分隔符（如段落分隔符 \n\n）分割出来的单元本身就比 chunk\_size 大，算法会尝试用下一个级别更低的分隔符（如句子分隔符 \n）对这个过大的单元进行再分割 16。然而，如果一个不可再分的语义单元（例如，一个无法在不破坏词义的前提下通过当前分隔符列表进一步分割的长句子）其自身长度就超过了 chunk\_size，但又不足以触发更细粒度的分割（或者更细粒度的分割会导致块过小而失去意义），那么这个单元可能会被保留为一个单独的块，即使它略微超出了 chunk\_size 的设定。这意味着 chunk\_size 在递归分块中更像是一个“软性”的指导目标，而非一个“硬性”的绝对上限。开发者在使用此方法时，应理解其在保持语义单元完整性和严格遵守块大小限制之间的这种内在权衡。

## III. 内容感知分块技术

与基础分块技术相比，内容感知分块技术更深入地考虑文本的内在结构和语义信息，旨在创建更自然、更符合人类阅读习惯的文本块。这类技术主要包括基于自然语言边界的句子切分和基于意义相似性的语义分块。

### A. 句子切分：利用自然语言边界

#### 1. 工作原理

句子切分（Sentence Splitting）是一种基于文本自然语言结构的分块方法，它以句子为基本单位进行文本分割 8。该方法利用句子结束符（如句号、问号、感叹号）作为切分点，将大段文本分解成一系列独立的句子或由多个连续句子组成的文本块 14。其核心目标是保持每个块内信息的完整性和单个思想或陈述的语义连贯性。

#### 2. 优点

句子切分的主要优点在于它能够很好地维护文本的句子结构，从而产生比固定大小字符或token分割更连贯的文本块 14。这种方法通常能更好地保留文本的语义完整性 14。此外，对于那些针对句子级别内容进行优化的嵌入模型（embedding models），基于句子的分块能够提供更匹配的输入，从而可能提升嵌入质量和下游任务的性能 1。

#### 3. 缺点

尽管句子切分在保持语义方面有优势，但也存在一些局限性。首先，自然语言中句子的长度差异很大，这会导致生成的文本块大小不一致，给后续处理带来不便 14。特别长的句子如果单独成块，仍可能超出某些模型的处理限制或理想的块大小范围 14。其次，句子切分器的性能可能受到语言类型的影响，并且在处理缩写词、复杂标点符号或非标准文本时，可能会发生错误切分 18。最后，大多数基础的句子切分器缺乏对跨句子上下文的深层语义理解，可能导致跨越多个句子的连贯信息在分块后被割裂 18。

#### 4. 使用 NLTK 和 spaCy 实现 (Native and via Langchain)

NLTK (Natural Language Toolkit) 和 spaCy 是两个广泛应用于自然语言处理的Python库，它们都提供了强大的句子切分功能。这些功能既可以直接在各自库中使用，也可以通过Langchain框架中集成的文本分割器来调用。

* **NLTK (Natural Language Toolkit):**
  + **原生用法 12:** NLTK 的 nltk.tokenize.sent\_tokenize 函数是进行句子切分的常用工具。  
    Python  
    import nltk  
    # nltk.download('punkt') # 首次使用需下载punkt分词器模型  
    from nltk.tokenize import sent\_tokenize  
      
    text = "This is the first sentence. This is the second sentence. Is this the third one?"  
    sentences = sent\_tokenize(text)  
    # sentences 将是:  
    for i, sentence in enumerate(sentences):  
     print(f"Sentence {i+1}: {sentence}")
  + **Langchain NLTKTextSplitter 1:** Langchain 提供了 NLTKTextSplitter 类，封装了NLTK的句子切分功能。  
    Python  
    from langchain\_text\_splitters import NLTKTextSplitter  
      
    text = "This is the first sentence. This is the second sentence. Is this the third one? NLTK helps split these."  
    text\_splitter = NLTKTextSplitter(language='english') # 可以指定语言  
    docs\_nltk = text\_splitter.split\_text(text)  
    # docs\_nltk 将是一个字符串列表，每个字符串是一个句子。  
    for i, doc\_text in enumerate(docs\_nltk):  
     print(f"NLTK Chunk {i+1}: {doc\_text}")
* **spaCy:**
  + **原生用法 24:** spaCy 通过其语言模型处理文本后，可以方便地迭代文档中的句子。  
    Python  
    import spacy  
      
    # nlp = spacy.load("en\_core\_web\_sm") # 加载spaCy模型，例如英文小模型  
    # text = "This is the first sentence. This is the second sentence. SpaCy is powerful."  
    # doc = nlp(text)  
    # sentences = [sent.text for sent in doc.sents]  
    # for i, sentence in enumerate(sentences):  
    # print(f"spaCy Sentence {i+1}: {sentence}")
  + **Langchain SpacyTextSplitter 1:** Langchain 的 SpacyTextSplitter 利用spaCy进行句子分割，并允许指定 chunk\_size 来将多个句子组合成一个块。  
    Python  
    from langchain\_text\_splitters import SpacyTextSplitter  
      
    text = "This is the first sentence. This is the second sentence. SpaCy provides robust sentence splitting via Langchain."  
    # `pipeline` 参数指定使用的spaCy模型。  
    text\_splitter = SpacyTextSplitter(pipeline='en\_core\_web\_sm', chunk\_size=100) # chunk\_size 可以将多个句子组合  
    docs\_spacy = text\_splitter.split\_text(text)  
    for i, doc\_text in enumerate(docs\_spacy):  
     print(f"spaCy Chunk {i+1}: {doc\_text}")
* **关于段落/章节分块的说明 19:** 虽然NLTK和spaCy主要提供句子级别的切分，但段落切分通常通过查找双换行符（\n\n）使用正则表达式来实现 19，或者通过组合连续的句子来完成。Langchain的RecursiveCharacterTextSplitter配合适当的分隔符（如 ["\n\n", "\n"]）通常是实现段落级别乃至句子级别分块的更直接方式。

### B. 语义分块：按意义分组

#### 1. 工作原理

语义分块（Semantic Chunking）是一种更高级的文本分割策略，它不依赖于固定的长度或句法分隔符，而是根据文本内容的语义相似性或主题的转变来划分文本块 1。该方法通常采用自然语言处理（NLP）技术，特别是利用嵌入模型（embedding models）来分析文本的语义内容 14。其核心思想是：将语义上紧密相关的句子或段落组合在一起形成一个块，而在文本主题发生显著变化的地方进行分割 10。具体实现上，这常常涉及到计算相邻文本单元（如句子或句子组）的嵌入向量之间的余弦相似度，当相似度低于某个预设阈值时，则认为是一个语义断点，应在此处分割 15。

#### 2. 优点

语义分块能够产生高度连贯的、与文档主题结构一致的文本块 14。这通常能带来更好的检索相关性，并使得LLM能够生成更准确、与上下文更契合的响应 1。由于它有效地保留了文本的语义完整性，因此在需要高度上下文理解的任务中表现出色 7。此外，通过将注意力集中在文档的相关部分，语义分块有助于优化对大型文档的处理效率，并能通过聚焦于最相关的信息来减少噪声和计算负荷 20。

#### 3. 缺点

语义分块的主要挑战在于其计算复杂度较高且处理速度相对较慢，因为它需要生成文本嵌入并进行相似度计算 7。其效果在很大程度上依赖于所使用的嵌入模型的质量和适用性；不同的嵌入模型对不同类型的文本和语义关系的捕捉能力各异 9。一个显著的问题是，标准的语义分块方法在处理包含非标准语义单元的文本（如数学公式、代码块、大段引文）时可能会遇到困难，因为这些元素的嵌入向量可能与周围的叙述性文本显著不同，从而导致不恰当的分割 23。此外，确定合适的相似度阈值（即语义断点）可能具有挑战性，需要根据具体数据和应用场景进行调整和实验 15。

#### 4. 常见用例

语义分块特别适用于那些对上下文理解、连贯性和主题连续性要求较高的任务 7。它是检索增强生成（RAG）系统的理想选择，因为高质量的语义块能显著提升检索结果的相关性和生成内容的准确性 20。此外，它也适用于分析那些具有清晰主题转换的文档，以及任何将保留文本语义完整性置于首要位置的应用场景 7。

#### 5. 使用 Langchain (SemanticChunker) 和 LlamaIndex (SemanticSplitterNodeParser) 实现

* **Langchain SemanticChunker 10:**
  + **解释:** SemanticChunker (位于 langchain\_experimental.text\_splitter 模块) 利用一个嵌入模型来分割文本。它会逐句将句子添加到一个块中，直到检测到一个“断点”，即句子之间或句子组之间的语义相似度显著下降。
  + **参数:** embeddings (一个嵌入模型实例)，breakpoint\_threshold\_type (断点阈值的类型，例如 "percentile", "standard\_deviation", "interquartile", "gradient")。
  + **代码示例 10:**  
    Python  
    # from langchain\_experimental.text\_splitter import SemanticChunker  
    # from langchain\_openai import OpenAIEmbeddings # 或如 [10] 中的 HuggingFaceEmbeddings  
      
    # text\_to\_split = "First topic sentence 1. First topic sentence 2. Second topic sentence A. Second topic sentence B."  
    # embeddings = OpenAIEmbeddings() # 替换为您偏好的嵌入模型  
      
    # semantic\_text\_splitter = SemanticChunker(  
    # embeddings,  
    # breakpoint\_threshold\_type="percentile" # 其他选项: "standard\_deviation", "gradient"  
    # )  
    # docs\_semantic = semantic\_text\_splitter.create\_documents([text\_to\_split])  
      
    # for i, doc in enumerate(docs\_semantic):  
    # print(f"Semantic Chunk {i+1}: {doc.page\_content}")
* **LlamaIndex SemanticSplitterNodeParser 1:**
  + **解释:** LlamaIndex 的 SemanticSplitterNodeParser 将文档分割成 Node 对象，每个节点代表一组语义相关的句子。它需要一个嵌入模型和一个句子分割器。
  + **参数:** embed\_model (嵌入模型)，breakpoint\_percentile\_threshold (余弦相异度的百分位阈值，超过此阈值则进行分割)，buffer\_size (用于评估语义相似性的句子分组数量)，sentence\_splitter (将文本分割成句子的可调用函数)。
  + **代码示例 21:**  
    Python  
    # from llama\_index.core.node\_parser import SemanticSplitterNodeParser  
    # from llama\_index.embeddings.openai import OpenAIEmbedding # 或其他嵌入模型  
    # from llama\_index.core import Document  
    # # LlamaIndex 早期版本使用 llama\_index.core.utils.globals\_helper.tokenizer 进行句子分割  
    # # 当前版本 (如 v0.10.17+) 默认使用 nltk.sent\_tokenize，或允许自定义 sentence\_splitter  
    # # from llama\_index.core.node\_parser.text\_splitter import SentenceSplitter # 这是一个更明确的句子分割器  
      
    # # 初始化嵌入模型  
    # embed\_model = OpenAIEmbedding()  
      
    # # LlamaIndex 的 SentenceSplitter  
    # # sentence\_splitter\_instance = SentenceSplitter(chunk\_size=1024) # chunk\_size 在此用于句子分割器内部逻辑，而非最终块大小  
    # # sentence\_splitter\_fn = sentence\_splitter\_instance.split\_text  
      
    # semantic\_node\_parser = SemanticSplitterNodeParser(  
    # embed\_model=embed\_model,  
    # breakpoint\_percentile\_threshold=95, # 默认: 95% 的相异度  
    # buffer\_size=1, # 默认: 逐句评估  
    # # sentence\_splitter=sentence\_splitter\_fn # 可以传入自定义的句子分割函数  
    # )  
      
    # documents =  
    # nodes = semantic\_node\_parser.get\_nodes\_from\_documents(documents)  
      
    # for i, node in enumerate(nodes):  
    # print(f"LlamaIndex Semantic Node {i+1}: {node.get\_content()}")
  + 21 提到：“语义分块器并非使用固定的块大小来切分文本，而是利用嵌入相似性自适应地在句子之间选择断点。”

内容感知分块技术，尤其是语义分块，其有效性高度依赖于所用嵌入模型的质量和特性 1。然而，当前的嵌入模型在处理非标准叙述性文本，如代码、数学公式或大段直接引语时，表现出一定的“语义脆弱性” 23。这些特殊类型的文本内容在嵌入空间中的表示可能与它们周围的常规文本有显著差异，导致语义分块算法基于这些差异做出错误的分割决策，例如将本应属于同一上下文的公式与其解释分割开。这意味着，仅仅依赖通用的语义分块方法可能不足以应对包含多样化内容类型的复杂文档。为了克服这一局限，可能需要采用针对特定领域优化的嵌入模型，或者发展出如“语义双通道合并”（semantic double-pass merging）这类更为复杂的混合策略，以便更鲁棒地处理这类混合内容 23。语义分块的质量与其底层嵌入模型对特定文本类型的表征能力直接相关。

观察分块技术的发展历程，从简单的固定大小或基于基本句法规则的切分，到递归分块中隐含的对段落等语义单元的层级化尊重，再到真正的语义分块明确运用意义进行划分，这一过程清晰地展示了分块方法从纯粹的结构或句法分割向基于语义内容的分割的演进趋势 2。这种演进与自然语言处理领域更广泛的进步——即越来越强调对“意义”的理解——是一致的。这一趋势预示着未来的分块创新将更侧重于细致入微的语义理解，而非任意的或纯粹的结构性断裂，甚至可能更多地利用大型语言模型自身来进行分块决策，正如在后续将讨论的“代理分块”中所见。这种转变意味着，要获得高质量的文本块，越来越需要更深层次的语言理解能力，从而推动了那些能够真正辨识主题边界和上下文关系的方法的发展。

## IV. 针对结构化和半结构化数据的专用分块技术

当处理具有明确结构或半结构（如Markdown、HTML、代码、LaTeX等）的文档时，采用通用的分块方法可能无法充分利用文档的内在组织信息。因此，发展出了一系列特定文档格式的分块器，它们能够解析这些格式特有的结构元素，从而实现更精准和符合逻辑的文本分割。

### A. 特定文档分块器 (Markdown, HTML, 代码, LaTeX)

#### 1. 工作原理

特定文档分块器（Document-Specific Chunkers）的核心原理是理解并利用特定文档格式的固有结构来指导分块过程 7。与依赖通用分隔符（如换行符或空格）或语义相似性的方法不同，这些分块器会解析文档中的结构化标记，例如Markdown的标题（#, ##等）、HTML的标签（<h1>, <p>, <li>等）、代码中的函数或类定义，或者LaTeX的命令和环境（如\section, \subsection）10。它们将这些结构元素识别为自然的分割边界，旨在最大程度上保留文档作者最初设定的逻辑组织和内容连贯性 7。

#### 2. 优点

这类分块器的主要优点在于能够精确地保留文档的逻辑结构和作者的组织意图 7。对于结构化文档而言，这通常能更有效地维持每个块内部的上下文完整性 7。由于结构标记本身往往就蕴含了语义分组的信息（例如，一个Markdown的二级标题下的内容通常围绕一个特定子主题展开），因此这种方法能够自然地将语义相关的文本组织在一起，从而在处理特定格式的文档时，对于下游任务（如信息检索或摘要生成）可能比通用分块方法更为有效 16。

#### 3. 缺点

特定文档分块器的主要局限性在于其专用性：它们仅适用于其设计所针对的特定文档格式，对于纯粹的非结构化文本或其他非标准格式的文档则不适用 7。此外，要稳健地解析某些复杂格式（如LaTeX或含有嵌套结构的HTML），可能需要相当复杂的解析器，其开发和维护成本较高。

#### 4. 常见用例

这些分块器广泛应用于处理以下类型的文档：

* **Markdown文档**：如软件的README文件、技术文档、笔记等，可以根据标题层级进行分割 10。
* **HTML文档**：用于从网页中提取和组织内容，可以基于HTML标签（如标题、段落、列表项）进行分块 14。
* **源代码文件**：如Python、JavaScript等，可以按函数、类或逻辑代码块进行分割，便于代码理解和分析 10。
* **LaTeX文档**：常用于学术论文和书籍，可以根据章节、小节等LaTeX结构命令进行分块 12。

#### 5. 使用 Langchain 和 LlamaIndex 实现

* **Langchain:** Langchain提供了一系列针对特定文档格式的文本分割器。
  + MarkdownTextSplitter / MarkdownHeaderTextSplitter 1: 这些类能够根据Markdown的标题（headers）来分割文本。MarkdownHeaderTextSplitter 尤其适合根据标题层级创建结构化的块，并将标题信息存储在元数据中。  
    Python  
    from langchain\_text\_splitters import MarkdownHeaderTextSplitter  
      
    markdown\_text = """  
    # Main Title  
    This is some text under the main title.  
    ## Subheading 1  
    Content for subheading 1.  
    ### Sub-subheading A  
    Details for A.  
    ## Subheading 2  
    Content for subheading 2.  
    """  
    headers\_to\_split\_on = [  
     ("#", "Header 1"),  
     ("##", "Header 2"),  
     ("###", "Header 3"),  
    ]  
    markdown\_splitter = MarkdownHeaderTextSplitter(headers\_to\_split\_on=headers\_to\_split\_on)  
    md\_header\_splits = markdown\_splitter.split\_text(markdown\_text)  
      
    # for split in md\_header\_splits:  
    # # 每个split包含page\_content和metadata (其中包含捕获的header信息)  
    # print(f"Header: {split.metadata}, Content: {split.page\_content[:30]}...")
  + HTMLHeaderTextSplitter / HTMLSectionSplitter 16: 用于根据HTML文档中的标题标签或其他特定标签来解析和分割内容。
  + PythonCodeTextSplitter (以及针对其他语言如JavaScript的JSFrameworkTextSplitter，和LatexTextSplitter) 10: 这些分割器尝试沿着特定编程语言的语法结构（如函数、类定义）或LaTeX的排版元素进行文本分割。  
    Python  
    # from langchain\_text\_splitters import PythonCodeTextSplitter  
    # python\_code = "def hello():\n print('Hello')\n\nclass MyClass:\n def \_\_init\_\_(self):\n pass"  
    # python\_splitter = PythonCodeTextSplitter(chunk\_size=100, chunk\_overlap=0) # chunk\_size和overlap在此处可能作用于代码块内部的进一步细分  
    # python\_docs = python\_splitter.create\_documents([python\_code])  
    # # python\_docs 会包含按Python语法结构（如函数、类）分割的块
* **LlamaIndex:** LlamaIndex也提供了处理特定文档格式的节点解析器（Node Parsers）。
  + MarkdownNodeParser 1: 与Langchain的Markdown分割器类似，它解析Markdown文本以创建结构化的节点（Nodes）。
  + LlamaIndex还支持多种文件阅读器（File Readers），这些阅读器本身就能理解某些文档结构（例如 PDFReader 26）。这些阅读器可以与节点解析器（如 TopicNodeParser 或 SemanticSplitterNodeParser）结合使用，以实现更高级的结构化数据处理 26。
  + LlamaIndex与Docugami的集成提供了一种智能分块方案，它能将文档分解为层级化的语义XML树，并包含语义注解和元数据，这有助于提升数据提取的准确性 26。

专用分块器通过利用文档的显式结构（如Markdown的标题、HTML的标签、代码的函数定义）来实现文本分割，这实际上带来了一种“语义结构”上的优势 7。文档的这些显式结构标记往往是作者用来组织思想和内容的，因此它们本身就承载了语义分组的意图。例如，HTML或Markdown中<h2>标签下的内容通常都围绕一个特定的子主题展开。当分块器沿着这些结构元素进行分割时，它们实际上是在执行一种“代理”式的语义分块，利用了文档自身的组织逻辑来实现语义连贯性，而无需依赖复杂的嵌入模型进行语义分析。这使得它们在处理其目标文档类型时，既高效又有效。

然而，特定文档分块器的强大功能也伴随着其固有的局限性。它们高度依赖于文档的格式规范性，对于非标准或混合格式的文档处理能力有限 7。现实世界中的数据往往是复杂和多样的，一个文档可能包含多种内容类型，例如一个Markdown文件内嵌了代码片段和自由文本段落。在这种情况下，单一的专用分块器可能无法最佳地处理文档的所有部分。这就引出了对更灵活、更鲁棒的混合分块策略的需求，这些策略能够智能地识别内容类型并切换或组合不同的分块方法。例如，“文本分块学习器混合模型（MoC）”框架就指向了这个方向，它通过一个路由器来动态选择最适合当前文本片段的专用分块器 5。这预示着未来的分块技术可能不再是单一方法的应用，而是一个能够根据具体情境动态调整的、适应性更强的策略组合。

## V. 高级分块策略和新兴概念

随着大型语言模型应用的深入发展，对文本分块技术的要求也日益提高，催生了一系列高级策略和新兴概念。这些方法试图克服传统分块技术的局限性，以期在上下文保留、语义理解和处理效率之间达到更优的平衡。

### A. 代理分块：LLM 驱动的切分

代理分块（Agentic Chunking）是一种前沿且尚处于实验阶段的文本分割方法，它利用大型语言模型（LLM）自身的能力来决定如何最恰当地切分文档 10。在这种模式下，LLM扮演一个“代理”的角色，通过分析文本内容的语义含义、内在结构（如段落类型、标题、指令序列等）以及当前任务的具体上下文，来智能地确定分块的边界 10。这种方法的核心在于模拟人类在处理和理解长文档时的推理过程，以期实现更符合逻辑和语义的分割 10。例如，LumberChunker便是利用LLM来识别连续段落中内容开始发散点的实践 5。

* **方法：** 其分割是基于LLM对文本的理解和特定任务的上下文，由人工智能驱动完成的 13。
* **潜在优势：** 对于结构异常复杂或非标准化的文档，传统基于规则或简单语义的方法可能难以奏效，而代理分块凭借LLM的理解能力，有望实现高度精细化和上下文最优化的分块结果。
* **潜在劣势：**
  + 计算成本高昂，因为分块决策本身就需要LLM进行推理调用。
  + 结果可能难以预测，且需要大量的提示工程（prompt engineering）和细致调整才能达到理想效果 13。
  + 目前仍处于探索和实验阶段，成熟度和普适性有待验证 10。
* **实现（概念性，参考 13）：**  
  Python  
  # from langchain.chat\_models import ChatOpenAI  
  # from langchain.prompts import PromptTemplate  
  # from langchain.chains import LLMChain  
  # #... 此处需要设置LLM以及用于指导分块的提示词...  
  # # 代理将接收文本，并根据指令决定如何分割。

### B. 块重叠在上下文保留中的作用

块重叠（Chunk Overlap）是一种在分块时常用的辅助技术，它指的是在一个块的末尾重复出现前一个块的一小部分文本内容，而在下一个块的开头也包含当前块末尾的一部分文本 2。

* **机制：** 这种重叠机制确保了如果一段关键信息或一个语义连接点恰好位于两个块的分割边界处，那么这部分信息至少能在一个完整的块中被捕获并传递给LLM 2。
* **益处：** 它显著降低了因文本分割而丢失跨块上下文的风险，尤其对于叙事性内容、常见问题解答（FAQ）、操作指南或知识库等需要保持上下文连续性的文本类型非常有用 1。
* **优化考量 1:**
  + **重叠大小：** 通常建议重叠量为块大小的10-20% 14。理想的重叠大小取决于文本的信息密度和具体类型。
  + **必要性：** 并非所有情况都需要重叠。应避免不必要的重叠，因为它会增加处理时间、存储需求，并可能在检索结果中引入重复内容 1。
  + **可变重叠（高级技巧，1）：** 一种更高级的技术是根据内容动态调整重叠量，例如对信息密集的叙述性内容使用较大的重叠，而对结构简单的部分则使用较小或不使用重叠。

### C. 优化块大小：一个平衡之举

确定最佳的块大小（Chunk Size）是分块策略中的一个核心挑战，它需要在确保信息精确性（通常倾向于较小块）和保留充分上下文（通常倾向于较大块）之间取得精妙的平衡 9。

* **核心挑战：** 过小的块可能导致上下文信息不足，使得模型难以理解完整意义或做出准确判断，如同管中窥豹 1。而过大的块不仅可能超出LLM的token处理上限，还可能因为包含过多不相关信息而稀释焦点，增加计算成本，甚至导致关键信息被忽略 1。正如一位开发者所遇到的问题：“一个大问题是机器人无法总结一本书或任何大型文档，因为它一次只能看到1-2个块” [41 (nopsled)]，这凸显了小块在处理需要广泛上下文的任务时的局限性。
* **影响因素 1:**
  + **LLM的Token限制：** 这是最硬性的约束。文本块必须能够适应模型的上下文窗口大小，理想情况下还应为提示词和模型的响应预留空间 1。
  + **文本复杂度：** 更复杂的文本可能需要更大的块来保留完整的意义，或者反而需要更小、更集中的块来精确捕捉特定细节。
  + **应用目标：**
    - 问答系统：倾向于较小、上下文丰富、能被快速精确检索的块 1。
    - 文档摘要：倾向于较大的块，以捕捉更广泛的主题和内容流 1。
    - 语义搜索：需要能够保留主题边界、富含关键词的块 1。
  + **嵌入模型的特性：** 不同的嵌入模型对块大小的偏好不同。例如，一些句子转换器（sentence transformers）模型在处理较短的块（如单个句子）时效果更佳，而像 text-embedding-ada-002 这样的模型则设计用于处理包含数百token的较大文本块 9。
* **通用指南：**
  + 从一个基准值开始，通过实验进行迭代优化。没有一刀切的完美尺寸，测试是关键 1。

### D. 简述：后期分块和上下文检索

后期分块（Late Chunking）和上下文检索（Contextual Retrieval）是针对检索增强生成（RAG）系统提出的高级技术，旨在克服传统（早期）固定大小分块方法容易造成的上下文碎片化问题 22。

* **后期分块 15:**
  + **过程：** 首先对整个文档进行token级别的嵌入。然后，将这些token嵌入序列分割成块。最后，对每个嵌入块应用均值池化（mean pooling）得到最终的块嵌入向量。
  + **目标：** 在进行块嵌入定义之前，保留文档完整的上下文信息。
  + **优点：** 计算效率较高，能适应多种长上下文嵌入模型，无需额外训练。
  + **缺点：** 与更复杂的上下文增强方法相比，可能在相关性和完整性方面有所牺牲。
* **上下文检索 22:**
  + **过程：** 在初步的文档分割之后，利用LLM为每个块生成来自整个文档的附加上下文或摘要，从而丰富每个块的信息量，即使是被分割的片段也能保持对整体内容的感知。例如，如果一个块是“公司营收较上一季度增长3%”，上下文增强会补充是哪个公司、哪个时间段。这通常还结合了排序融合（rank fusion，如结合BM25等稀疏向量检索）和重排序（reranking）步骤。
  + **目标：** 更有效地保留语义连贯性。
  + **优点：** 语义连贯性更好，通过上下文丰富能提升检索结果。
  + **缺点：** 需要更多的计算资源，因为涉及LLM调用来进行上下文生成和重排序。

代理分块策略将LLM的角色从仅仅处理预设的文本块，提升到了参与分块决策本身的高度 10。这代表了一种从预定义算法向学习型、动态型方法的转变。LLM在此过程中运用其推理能力来优化文本的分割方式，以适应特定的下游任务。这种“元级别”的分块，即“对分块过程进行分块”，潜力巨大，尤其是在处理传统方法难以应对的复杂或非典型文档结构时。然而，这也带来了与LLM推理相关的可控性、可解释性和成本等方面的挑战。它实质上是在“手工定制”的分块逻辑与“习得”的分块逻辑之间进行权衡。

与此同时，诸如“后期分块”和“上下文检索”等技术的出现，表明了高级分块策略正日益超越单纯的文本分割范畴，与嵌入向量的生成方式以及在RAG流程中如何向检索器和LLM提供上下文等环节深度融合 22。分块不再是一个孤立的预处理步骤，而已成为检索和上下文增强策略中不可或缺的一环。分块决策受到嵌入策略的影响，而检索质量又因块的形成和上下文化方式得到提升。这意味着RAG系统的优化需要一个更整体的设计视角，将分块、嵌入和检索策略协同考虑，而非割裂对待。

## VI. 实践指南：选择和实施正确的分块策略

选择和实施合适的分块策略是构建高效LLM应用的关键步骤。由于不存在一种万能的策略，开发者需要综合考虑多种因素，并通过实验来确定最适合自身应用场景的方法。

### A. 影响选择的因素

在决定采用何种分块技术时，应仔细评估以下几个关键方面 1：

1. **内容类型和结构:**
   * **非结构化文本（如散文、文章）：** 递归分块或基于句子的分块通常是良好的起点。如果对质量要求较高且计算成本可接受，可以考虑语义分块 1。
   * **半结构化文本（如Markdown、HTML）：** 应优先使用特定于文档结构的分块器（例如，MarkdownHeaderTextSplitter），它们能够利用固有的结构信息（如标题）进行分割 1。
   * **结构化数据（如表格、代码）：** 需要专门的分块器，例如按行分割表格数据，或按函数/类分割代码 14。
   * **缺乏自然断点的文本：** 在这种情况下，固定大小分块可能是必要的，尽管它可能牺牲一些语义连贯性 1。
2. **应用目标 / 下游任务:**
   * **问答（Q&A）：** 受益于较小、专注且上下文丰富的块，以便快速、精确地检索到答案 1。
   * **摘要生成：** 更适合使用较大的块，因为它们能捕捉更广泛的主题覆盖和内容流 1。
   * **语义搜索：** 需要能够保留主题边界、富含关键词的块 1。
   * **对话式AI：** 块需要能够支持上下文的连续性和对话的流畅性 14。
3. **LLM的约束条件:**
   * **Token限制：** 文本块的大小必须适应所用LLM的上下文窗口限制 1。使用token计数工具（如tiktoken）来估算和控制块的token数量至关重要 1。
   * **所用嵌入模型：** 块的大小和类型应与所选嵌入模型的特性相匹配 9。例如，句子转换器（sentence transformers）通常更适合处理较短的块，而像OpenAI的text-embedding-ada-002这样的模型则能更好地处理包含数百token的较大块。
4. **计算资源与延迟要求:**
   * 固定大小分块以及基础的递归分块或句子切分通常速度更快，对计算资源的消耗也更少。
   * 相比之下，语义分块和代理分块由于涉及到嵌入生成或LLM调用，计算成本更高，延迟也可能更大 7。
5. **上下文保留与粒度需求之间的平衡:**
   * 如果跨块边界的上下文信息至关重要，应考虑使用块重叠（chunk overlap）1。
   * 需要在小而精确的块（可能丢失部分上下文）与大而富含上下文的块（可能稀释焦点或超出限制）之间做出权衡 9。
6. **测试与迭代:**
   * 由于没有一种策略能完美适应所有情况，因此必须对不同的分块方法和参数进行测试，并根据实际性能指标（如响应相关性、处理速度、用户满意度等）进行优化 1。这是一个持续迭代的过程。

### B. 比较概览表

为了更直观地比较各种主要的分块策略，下表总结了它们的原理、关键优缺点、典型用例以及在Langchain等框架中的示例实现：

| **策略 (Strategy)** | **原理 (Principle)** | **关键优势 (Key Advantages)** | **关键劣势 (Key Disadvantages)** | **典型用例 (Typical Use Cases)** | **主要库/类 (Langchain 示例)** |
| --- | --- | --- | --- | --- | --- |
| **固定大小 (Fixed-Size)** | 按字符/token计数的统一分段。 | 简单，快速，统一。 7 | 忽略语义，可能打断句子/上下文。 3 | 统一数据，无自然断点。 1 | CharacterTextSplitter 10 |
| **递归字符 (Recursive Character)** | 按层级分隔符（段落、句子、词）迭代分割。 | 适应结构，比固定大小更好地保留上下文。 7 | 块大小不保证，依赖分隔符。 7 | 通用非结构化文本。 17 | RecursiveCharacterTextSplitter 10 |
| **句子切分 (Sentence Splitting)** | 按句子边界分割。 | 保留句子完整性，适合句子嵌入器。 12 | 块大小可变，长句可能是问题。 14 | 当单个句子意义关键时。 1 | NLTKTextSplitter, SpacyTextSplitter 12 |
| **语义分块 (Semantic Chunking)** | 使用嵌入按语义相似性分组。 | 高度连贯，与主题一致，检索效果好。 1 | 计算昂贵，依赖嵌入质量，难以处理代码/公式。 7 | RAG，上下文敏感任务。 7 | SemanticChunker (Langchain Exp.) 10 |
| **特定文档 (Document-Specific)** | 解析结构（Markdown标题，HTML标签，代码函数）。 | 保留逻辑结构，“免费的”语义连贯性。 7 | 格式特定，不适用于纯文本。 7 | 结构化文档（Markdown, HTML, 代码）。 12 | MarkdownHeaderTextSplitter, PythonCodeTextSplitter 12 |
| **代理分块 (Agentic Chunking)** | LLM根据内容和任务决定分块边界。 | 潜在的高度优化和自适应。 10 | 实验性，昂贵，不可预测。 10 | 复杂，非标准文档。 | 自定义LLM链 13 |

选择最佳分块策略的过程本质上是一个迭代和实验的过程 1。正如 11 所指出的，“确定最佳分块策略可以通过迭代过程完成，该过程涉及评估LLM在每种分块策略或块大小上的性能。” 这意味着在数据准备流程中，应设计足够的灵活性，以便能够方便地更换分块方法和调整参数，并建立稳健的评估框架来衡量这些选择对下游应用性能的实际影响。这提示我们，LLM的数据操作（MLOps for LLMs）中，对数据版本、分块参数的版本控制以及实验结果的追踪变得日益重要。

此外，一个常被忽视但至关重要的实践细节是分词器（tokenizer）的选择对基于token的分块的精确控制影响 1。LLM和嵌入模型通常有其特定的分词器，同一段文本在不同分词器下产生的token数量可能存在差异 31。如果分块时依据的是字符数或一个通用的分词器，那么对于目标模型而言，实际的token数可能与预期不符，导致块过大或过小。因此，为了实现精确的基于token的分块，强烈建议使用与最终将处理这些块的目标模型（LLM或嵌入模型）相同或高度兼容的分词器（例如，针对OpenAI模型使用tiktoken）来计量块大小。Langchain等框架提供的辅助函数（如CharacterTextSplitter.from\_huggingface\_tokenizer或from\_tiktoken\_encoder）正是为了方便实现这一点。

## VII. 结论与未来展望

文本分块作为大型语言模型（LLM）数据准备流程中的一个基础且关键的环节，其重要性不言而喻。有效的文本分块不仅是克服LLM输入长度限制的技术需求，更是确保模型能够准确理解上下文、高效检索信息并生成高质量响应的基石。

### A. 关键分块技术和框架回顾

本报告系统地梳理了当前主流的实用分块技术。从基础的**固定大小分块**和**递归分块**，到更侧重内容理解的**句子切分**和**语义分块**，再到针对特定格式的**特定文档分块器**，以及前沿的**代理分块**概念，每种技术都有其独特的原理、优势、劣势和适用场景。固定大小分块简单直接，但可能破坏语义；递归分块试图通过层级分隔符改善上下文保留；句子切分利用自然语言边界；语义分块则追求深层次的意义聚合；而特定文档分块器则能充分利用已有结构。

开源框架如 **Langchain** 和 **LlamaIndex** 极大地简化了这些分块技术的实现和集成，它们提供了丰富的文本分割器（Text Splitters / Node Parsers）以及与嵌入模型、向量存储和LLM本身交互的工具链。同时，像 **NLTK** 和 **spaCy** 这样的经典NLP库也为句子切分等基础操作提供了坚实支持。

最终，选择何种分块策略，以及如何配置块大小、重叠等参数，需要在管理LLM的token限制、最大化上下文保留、以及与具体应用目标（如问答、摘要、搜索）对齐之间进行细致的权衡。

### B. 发展格局：新研究与评估方法

文本分块领域正经历着从简单的基于规则的方法向更复杂、更具语义感知能力，乃至由LLM驱动的动态策略演进的过程 5。

* **技术趋势与高级方法：** 研究者们正在积极探索如**后期分块（Late Chunking）**、\*\*上下文检索（Contextual Retrieval）**以及各种形式的**自适应或动态分块（Adaptive/Dynamic Chunking）\*\*等高级技术，以期更有效地处理长文本并保留全局上下文 1。这些方法试图将分块过程与嵌入生成和检索机制更紧密地结合起来。
* **评估方法的革新：** 传统上，分块策略的有效性多是通过其对下游任务（如RAG系统的问答准确率）的间接影响来评估的 5。然而，这种方式难以隔离分块本身的质量。近期，研究领域开始出现直接针对分块质量本身的评估指标和框架：
  + **MoC（Mixtures of Text Chunking Learners）框架**引入了“边界清晰度（Boundary Clarity）”和“块内聚性（Chunk Stickiness）”双指标，并强调了将LLM集成到分块过程中的必要性，以克服传统和语义分块在处理复杂上下文细微差别时的局限性 5。
  + **HOPE（Holistic Passage Evaluation）度量标准**则从块的内在属性（概念统一性）、外在属性（语义独立性）以及块与文档的连贯性三个层面进行评估 32。该研究的一个有趣发现是，与传统认知不同，块的“概念统一性”对RAG性能的影响可能不如“语义独立性”那么显著，后者被证明对系统性能至关重要 32。

这种向直接评估的转变标志着分块技术作为一个独立研究领域的成熟。然而，这也带来了一个“先有鸡还是先有蛋”的问题：我们是应该基于内在属性（如HOPE度量中的语义连贯性）来定义“好”的块，还是应该通过它们在下游任务中的效用来定义？最有效的评估方法可能需要结合内在质量度量和基于任务的外部性能评估，以确保理论上“好”的块在实践中也同样有用。

* **未来展望：**
  + **更强的自适应性和上下文感知：** 未来的分块策略将更加智能，能够根据文本内容、用户查询意图和任务需求动态调整。
  + **与嵌入和检索的深度融合：** 分块将不再是孤立的预处理步骤，而是与嵌入表示和信息检索过程协同设计、共同优化。
  + **标准化基准和评估协议：** 随着研究的深入，有望出现更广泛接受的用于比较不同分块方法的基准数据集和评估标准。
  + **LLM在分块中的核心作用：** LLM不仅作为处理分块后数据的消费者，也可能更多地参与到定义最佳分块策略的过程中，实现“智能体化”分块。

从一个更广阔的视角来看，复杂精妙的分块技术不仅仅是为了将数据适配模型的输入窗口，它更像是一种针对LLM的初步“知识蒸馏”或“信息压缩”过程 1。它并非简单地分割文本，而是战略性地预处理和结构化外部知识，使其能被LLM最有效地吸收和利用。每一个精心划分的块都成为原始文档中关键信息的浓缩表示，专为LLM的“消化系统”量身定制。这种观点将分块从一种技术上的必要操作，提升到了向LLM进行知识传递的战略组成部分，突显了在设计分块策略时，为最大化每块信息密度和相关性所付出的智力努力。

总之，文本分块技术是LLM应用开发中一个持续演进且充满挑战的领域。随着LLM能力的不断增强和应用场景的日益复杂化，对更智能、更高效、更易于评估的分块方法的需求也将持续增长。

#### Works cited

1. Chunking Strategy for LLM Application: Everything You Need to Know, accessed June 1, 2025, <https://aiveda.io/blog/chunking-strategy-for-llm-application>
2. Chunking Strategies for LLM Applications: The Complete Guide - GlobalNodes AI, accessed June 1, 2025, <https://globalnodes.tech/blog/chunking-strategy-for-llm-application/>
3. Financial Report Chunking for Effective Retrieval Augmented Generation - arXiv, accessed June 1, 2025, <https://arxiv.org/html/2402.05131v2>
4. Financial Report Chunking for Effective Retrieval Augmented Generation - arXiv, accessed June 1, 2025, <https://arxiv.org/html/2402.05131v3>
5. MoC: Mixtures of Text Chunking Learners for Retrieval-Augmented Generation System, accessed June 1, 2025, <https://arxiv.org/html/2503.09600v1>
6. Retrieval-Augmented Generation for Large Language Models: A Survey - arXiv, accessed June 1, 2025, <https://arxiv.org/html/2312.10997v5>
7. Chunking strategies in RAG: The quest for the perfect pieces, accessed June 1, 2025, <https://lftechnology.com/blog/chunking-strategies-RAG>
8. A Deep-Dive into Chunking Strategy, Chunking Methods, and Precision in RAG Applications, accessed June 1, 2025, <https://www.superteams.ai/blog/a-deep-dive-into-chunking-strategy-chunking-methods-and-precision-in-rag-applications>
9. Advanced Chunking Strategies for LLM Applications | Optimizing ..., accessed June 1, 2025, <https://dev-kit.io/blog/ai/advanced-chunking-strategies-for-llm-applications-optimizing-efficiency-and-accuracy>
10. Chunking strategies for RAG tutorial using Granite | IBM, accessed June 1, 2025, <https://www.ibm.com/think/tutorials/chunking-strategies-for-rag-with-langchain-watsonx-ai>
11. Maximizing LLM Performance with Effective Chunking Strategies for Vector Embeddings, accessed June 1, 2025, <https://vectorshift.ai/blog/maximizing-llm-performance-with-effective-chunking-strategies-for-vector-embeddings>
12. Vinija's Notes • NLP • Retrieval Augmented Generation - Vinija Jain, accessed June 1, 2025, <https://vinija.ai/nlp/RAG/>
13. Chunking Strategies for Optimizing Large Language Models (LLMs), accessed June 1, 2025, <https://myscale.com/blog/chunking-strategies-for-optimizing-llms/>
14. 5 Chunking Techniques for Retrieval-Augmented Generation (RAG), accessed June 1, 2025, <https://www.apxml.com/posts/rag-chunking-strategies-explained>
15. NLP • Retrieval Augmented Generation - aman.ai, accessed June 1, 2025, <https://aman.ai/primers/ai/RAG/>
16. Text splitters | 🦜️ LangChain, accessed June 1, 2025, <https://python.langchain.com/docs/concepts/text_splitters/>
17. RecursiveCharacterTextSplitter — LangChain 0.0.149 - Read the Docs, accessed June 1, 2025, <https://lagnchain.readthedocs.io/en/stable/modules/indexes/text_splitters/examples/recursive_text_splitter.html?highlight=RecursiveCharacterTextSplitter>
18. How to Chunk Text Data - A Comparative Analysis - Towards Data Science, accessed June 1, 2025, <https://towardsdatascience.com/how-to-chunk-text-data-a-comparative-analysis-3858c4a0997a/>
19. regex - Split very long text into paragraphs with nltk and python ..., accessed June 1, 2025, <https://stackoverflow.com/questions/54906645/split-very-long-text-into-paragraphs-with-nltk-and-python>
20. Semantic Chunking for RAG: Better Context, Better Results, accessed June 1, 2025, <https://www.multimodal.dev/post/semantic-chunking-for-rag>
21. Semantic Chunker - LlamaIndex, accessed June 1, 2025, <https://docs.llamaindex.ai/en/stable/examples/node_parsers/semantic_chunking/>
22. Reconstructing Context - arXiv, accessed June 1, 2025, <https://arxiv.org/html/2504.19754v1>
23. Chunking methods in RAG: comparison - BitPeak, accessed June 1, 2025, <https://bitpeak.com/chunking-methods-in-rag-methods-comparison/>
24. A Primer on Text Chunking and its Types - LanceDB Blog, accessed June 1, 2025, <https://blog.lancedb.com/a-primer-on-text-chunking-and-its-types-a420efc96a13/>
25. langchain-text-splitters: 0.3.8, accessed June 1, 2025, <https://python.langchain.com/api_reference/text_splitters/index.html>
26. [Question]: How to chunk text for structured LLM data extraction #18240 - GitHub, accessed June 1, 2025, <https://github.com/run-llama/llama_index/issues/18240>
27. [2503.09600] MoC: Mixtures of Text Chunking Learners for Retrieval-Augmented Generation System - arXiv, accessed June 1, 2025, <https://arxiv.org/abs/2503.09600>
28. [2504.19754] Reconstructing Context: Evaluating Advanced Chunking Strategies for Retrieval-Augmented Generation - arXiv, accessed June 1, 2025, <https://arxiv.org/abs/2504.19754>
29. Reconstructing Context: Evaluating Advanced Chunking Strategies for Retrieval-Augmented Generation - ResearchGate, accessed June 1, 2025, <https://www.researchgate.net/publication/391246731_Reconstructing_Context_Evaluating_Advanced_Chunking_Strategies_for_Retrieval-Augmented_Generation>
30. arxiv.org, accessed June 1, 2025, <https://arxiv.org/pdf/2504.19754>
31. Tokenizer - Hugging Face, accessed June 1, 2025, <https://huggingface.co/docs/transformers/main_classes/tokenizer>
32. A New HOPE: Domain-agnostic Automatic Evaluation of Text Chunking - arXiv, accessed June 1, 2025, <https://arxiv.org/html/2505.02171v1>
33. MoC: Mixtures of Text Chunking Learners for Retrieval-Augmented Generation System, accessed June 1, 2025, <https://www.researchgate.net/publication/389786403_MoC_Mixtures_of_Text_Chunking_Learners_for_Retrieval-Augmented_Generation_System>
34. [2505.02171] A New HOPE: Domain-agnostic Automatic Evaluation of Text Chunking - arXiv, accessed June 1, 2025, <https://arxiv.org/abs/2505.02171>
35. A New HOPE: Domain-agnostic Automatic Evaluation of Text Chunking - arXiv, accessed June 1, 2025, <http://arxiv.org/pdf/2505.02171>
36. arxiv.org, accessed June 1, 2025, <https://arxiv.org/pdf/2505.02171>
37. How to Chunk Text Data - A Comparative Analysis | Towards Data ..., accessed June 1, 2025, <https://towardsdatascience.com/how-to-chunk-text-data-a-comparative-analysis-3858c4a0997a>
38. Sentence segmentation using spaCy - YouTube, accessed June 1, 2025, <https://www.youtube.com/watch?v=NknDZSRBT7Y>
39. SemanticSplitterNodeParser - LlamaIndex v0.10.17, accessed June 1, 2025, <https://docs.llamaindex.ai/en/v0.10.17/api/llama_index.core.node_parser.SemanticSplitterNodeParser.html>
40. Semantic splitter - LlamaIndex, accessed June 1, 2025, <https://docs.llamaindex.ai/en/stable/api_reference/node_parsers/semantic_splitter/>
41. Chunking Strategies for LLM applications - General - Pinecone Community, accessed June 1, 2025, <https://community.pinecone.io/t/chunking-strategies-for-llm-applications/1067>