Strategy	Best For	Complexity	Context Preservation	Performance
Sliding Window	Medical records, long narratives	Low	⭐⭐⭐⭐⭐	Fast
Adaptive	Legal docs, contracts	Medium	⭐⭐⭐⭐	Medium
Entity-based	News, knowledge bases	High	⭐⭐⭐	Slow (NER required)
Topic-based	Research papers, articles	High	⭐⭐⭐⭐	Slow (embedding + clustering)
Hybrid	Technical docs, complex content	Very High	⭐⭐⭐⭐⭐	Slow
Task-aware	Multi-purpose systems	Medium	⭐⭐⭐⭐	Variable
HTML/XML	Web content, structured docs	Low	⭐⭐⭐⭐	Fast

RAG 2.0 : Advanced Chunking Strategies with Examples.

12 min read · Oct 2, 2025

Vishal Mysore Follow

Text chunking is a critical preprocessing step in RAG Based Large Language Model (LLM) applications. While basic strategies like fixed-size, recursive, semantic, document-based, and agentic chunking are well-established, there exist several advanced techniques that can dramatically improve retrieval accuracy, context preservation, and task-specific performance.

This comprehensive guide explores **9 advanced chunking strategies** with real-world examples, implementation insights, and visual diagrams to help you choose the right approach for your use case.

🧠 Ready to test what you've learned in this article? Try a quiz on EasyQZ after reading the article! 📚🎯

1. Sliding Window Chunking

Domain: Healthcare Patient Notes

Concept: Sliding Window Chunking creates overlapping chunks by moving a fixed-size window across the text with a specified overlap. This ensures continuity of context across chunk boundaries.

How It Works:

- Define a window size (e.g., 500 words)
- Define an overlap size (e.g., 100 words)
- Slide the window across the document, creating chunks that share common content

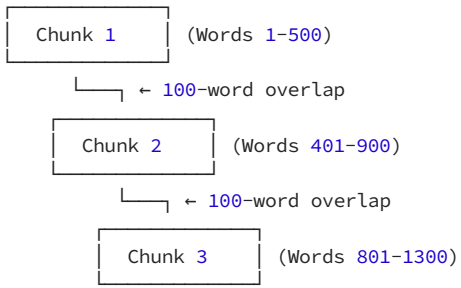
Example:

A 10,000-word patient record is chunked using a 500-word window with 100-word overlap:

```
Chunk 1: Words 1-500
  "Patient admitted on Jan 5 with acute respiratory distress...
   vital signs: BP 140/90, HR 88, stable condition..."

Chunk 2: Words 401-900
  "...vital signs: BP 140/90, HR 88, stable condition... [OVERLAP]
   began treatment protocol with corticosteroids..."
```

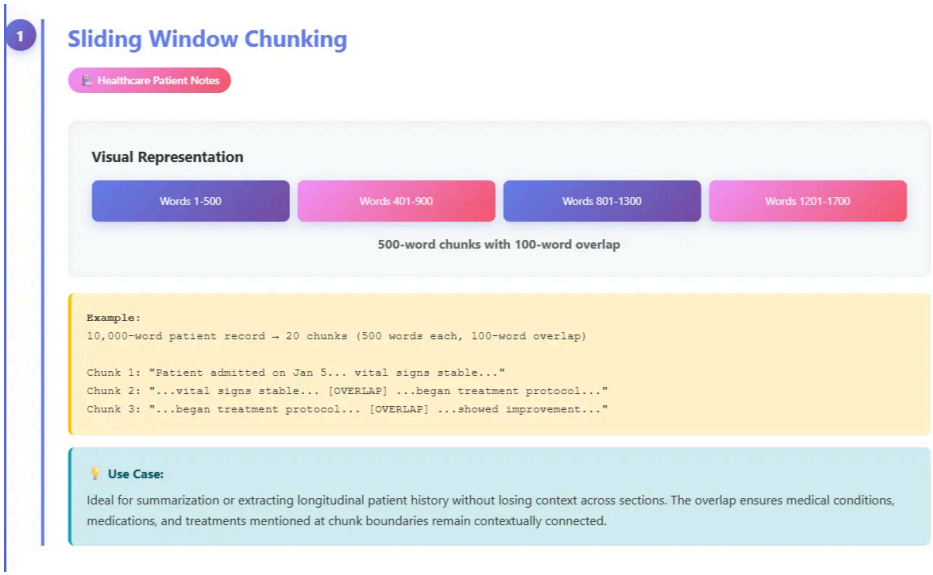

12/4/25, 11:21 AM RAG 2.0 : Advanced Chunking Strategies with Examples. | by Vishal Mysore | Medium

```
Chunk 3: Words 801-1300
  "...began treatment protocol with corticosteroids... [OVERLAP]
   patient showed improvement in oxygen saturation..."
```



Use Case: Ideal for summarization or extracting longitudinal patient history without losing context across sections. The overlap ensures medical conditions, medications, and treatments mentioned at chunk boundaries remain contextually connected, preventing information loss when tracking patient progress over time.

Implementation Tip: Adjust overlap based on context importance — use 20% overlap for dense medical records, 10% for general documents.



2. Adaptive Chunking

Domain: Legal Contracts

Concept: Adaptive Chunking creates variable-size chunks that respect natural document boundaries while staying within token limits. Rather than enforcing strict size limits, it adapts to the content structure.

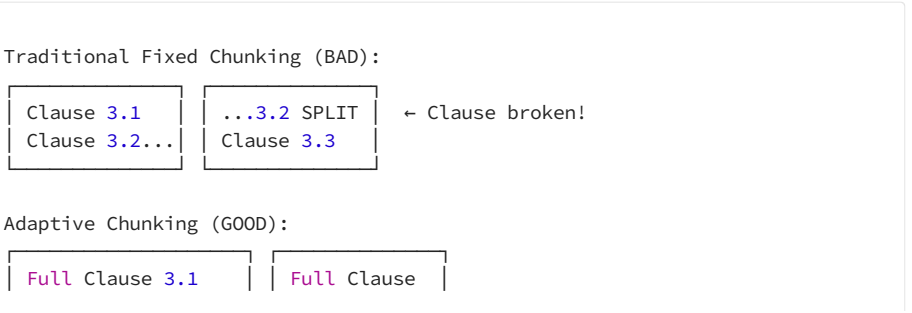
How It Works:

- Set a target token range (e.g., 800–1200 tokens)
- Identify natural breakpoints (clauses, sections, paragraphs)
- Create chunks that maximize token usage without splitting logical units

Example:

Legal contract with multiple sections:

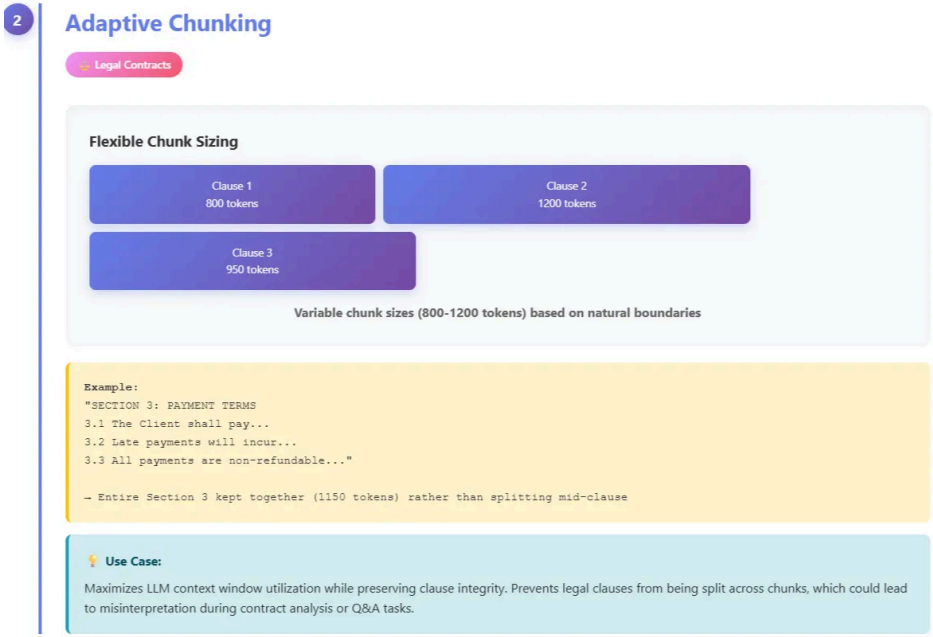
```
SECTION 3: PAYMENT TERMS (1,150 tokens)
├── 3.1 The Client shall pay the Contractor... (380 tokens)
├── 3.2 Late payments will incur a penalty... (290 tokens)
├── 3.3 All payments are non-refundable... (250 tokens)
└── 3.4 Payment disputes shall be resolved... (230 tokens)

→ Entire Section 3 kept together as ONE CHUNK (1,150 tokens)
  rather than splitting mid-clause

SECTION 4: CONFIDENTIALITY (950 tokens)
→ Kept as separate complete chunk
```



```
| Full Clause 3.2    | | 4.1          |
| Full Clause 3.3    | | Full Clause  |
└────────────────────┘ └──────────────┘
```

Use Case: Maximizes LLM context window utilization while preserving clause integrity. Prevents legal clauses from being split across chunks, which could lead to misinterpretation during contract analysis, compliance checking, or Q&A tasks. Essential for maintaining legal precision.

Implementation Tip: Use AST parsing or regex patterns to detect section boundaries (e.g., “SECTION X:”, numbered clauses).



3. Entity-Based Chunking

Domain: News Archives

Concept: Entity-Based Chunking groups sentences by the entities they mention, creating entity-centric chunks rather than sequential text blocks.

How It Works:

- Run Named Entity Recognition (NER) on the document
- Group sentences that mention the same entity
- Create separate chunks for each primary entity

Example:

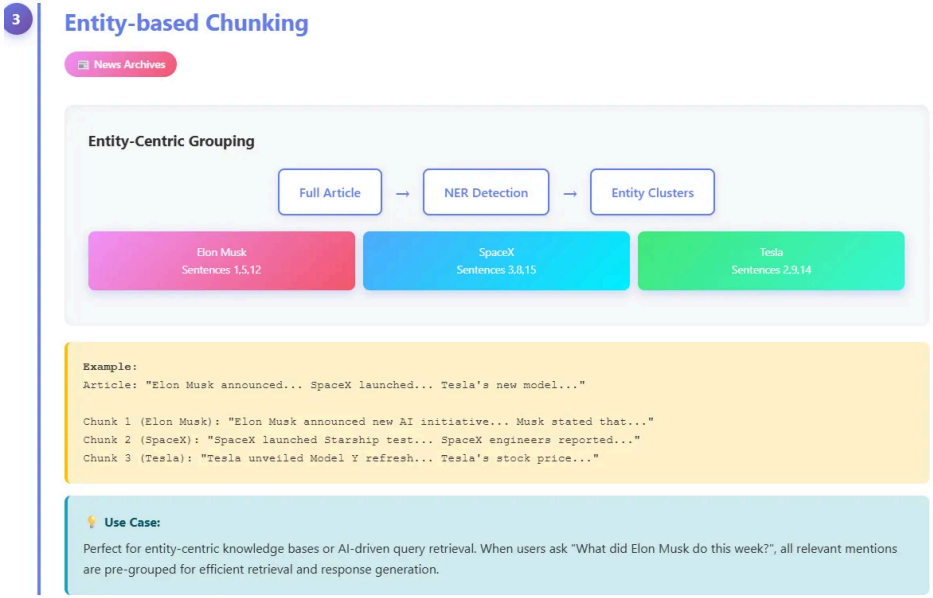
News article about tech industry:

```
Original Article:
[1] Elon Musk announced a new AI initiative yesterday.
[2] Tesla unveiled its Model Y refresh at the event.
[3] SpaceX launched its Starship test flight successfully.
[4] The billionaire stated his concerns about AI safety.
[5] Musk's companies continue to innovate.
[6] SpaceX engineers reported record performance.

Entity-Based Chunks:

Chunk 1 (Elon Musk):
  "Elon Musk announced a new AI initiative yesterday.
   The billionaire stated his concerns about AI safety.
   Musk's companies continue to innovate."

Chunk 2 (SpaceX):
  "SpaceX launched its Starship test flight successfully.
   SpaceX engineers reported record performance."

Chunk 3 (Tesla):
  "Tesla unveiled its Model Y refresh at the event."
```

```
Original Document → NER Detection → Entity Clustering
                                          ↓
┌──────────┐ ┌──────────┐ ┌──────────┐ ┌──────────┐
│  Elon    │ │ SpaceX   │ │  Tesla   │ │   AI     │
│  Musk    │ │          │ │          │ │  Safety  │
│ Chunk 1  │ │ Chunk 2  │ │ Chunk 3  │ │ Chunk 4  │
└──────────┘ └──────────┘ └──────────┘ └──────────┘
```

Use Case: Perfect for entity-centric knowledge bases or AI-driven query retrieval. When users ask “What did Elon Musk do this week?”, all relevant mentions are pre-grouped for efficient retrieval and response generation. Dramatically improves retrieval precision for entity-focused queries.

Implementation Tip: Use spaCy or Hugging Face NER models. Consider co-reference resolution to handle pronouns (e.g., “he” → “Elon Musk”).



4. Topic/Theme-Based Chunking

Domain: Research Papers

Concept: Topic-Based Chunking uses semantic similarity and clustering algorithms to group paragraphs by underlying themes, even if they’re not contiguous in the original document.

How It Works:

- Generate embeddings for each paragraph/section

- Apply clustering (K-means, HDBSCAN) to group similar content
- Create chunks from semantically related paragraphs

Example:

Research paper with 50 paragraphs:

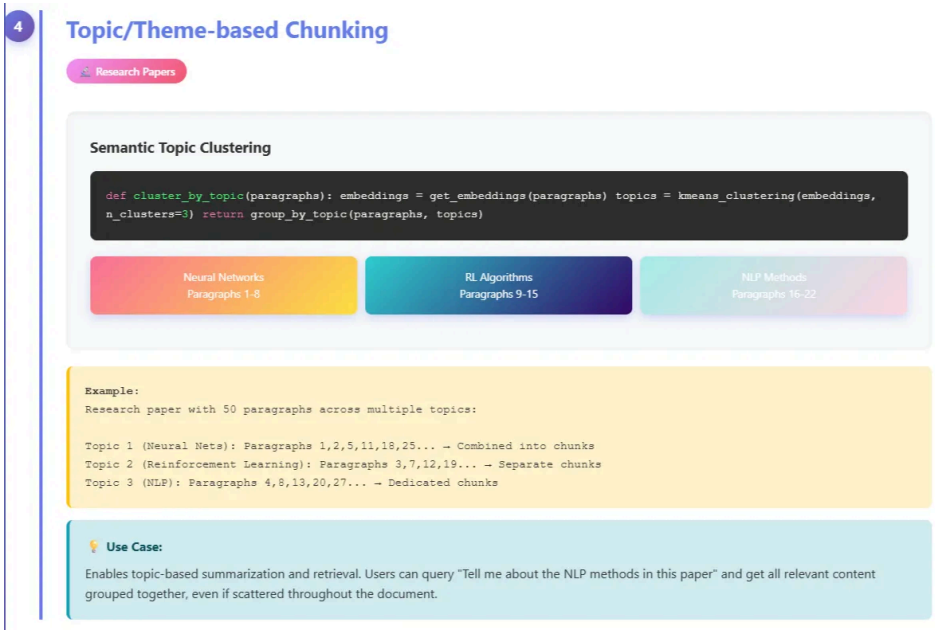
```
Original Order:
Para 1: Introduction to neural networks
Para 2: Dataset description
Para 3: Reinforcement learning background
Para 4: NLP preprocessing steps
Para 5: Neural network architecture details
...

Topic-Based Chunks:

Topic 1 - Neural Networks:
  Paragraphs [1, 5, 11, 18, 25, 33, 41]
  → Combined into coherent chunks about NN architectures

Topic 2 - Reinforcement Learning:
  Paragraphs [3, 7, 12, 19, 27, 35, 43]
  → All RL content grouped together

Topic 3 - NLP Methods:
  Paragraphs [4, 8, 13, 20, 28, 36, 44]
  → NLP preprocessing and techniques
```

```
Semantic Clustering Process:

Original Doc → Embed Paragraphs → K-Means Clustering
 (50 paras)      (Vectors)          (3 topics)
                                        ↓
                    ┌──────────────────┼──────────────────┐
                    │                  │                  │
                    ▼                  ▼                  ▼
            ┌──────────────┐   ┌──────────────┐   ┌──────────────┐
            │ Neural Nets  │   │ Reinforce-   │   │   NLP        │
            │              │   │ ment         │   │  Methods     │
            │ Paras:       │   │ Learning     │   │              │
            │ 1,5,11,18,   │   │              │   │ Paras:       │
            │ 25,33,41     │   │ Paras:       │   │ 4,8,13,      │
            │              │   │ 3,7,12,19,   │   │ 20,28,36     │
```

```
                                    27,35,43
```

Use Case: Enables topic-based summarization and retrieval. Users can query “Tell me about the NLP methods in this paper” and get all relevant content grouped together, even if scattered throughout the document. Excellent for literature review, research synthesis, and targeted information extraction.

Implementation Tip: Use sentence-transformers for embeddings, experiment with different clustering algorithms (K-means for known topic count, HDBSCAN for automatic detection).



5. Hybrid Chunking

Domain: Software Documentation

Concept: Hybrid Chunking combines multiple strategies in a pipeline to leverage the strengths of each approach.

How It Works:

- Apply multiple chunking strategies sequentially
- Each strategy refines the output of the previous one
- Common combinations: Recursive + Semantic + Entity-based

Example:

API documentation processing:

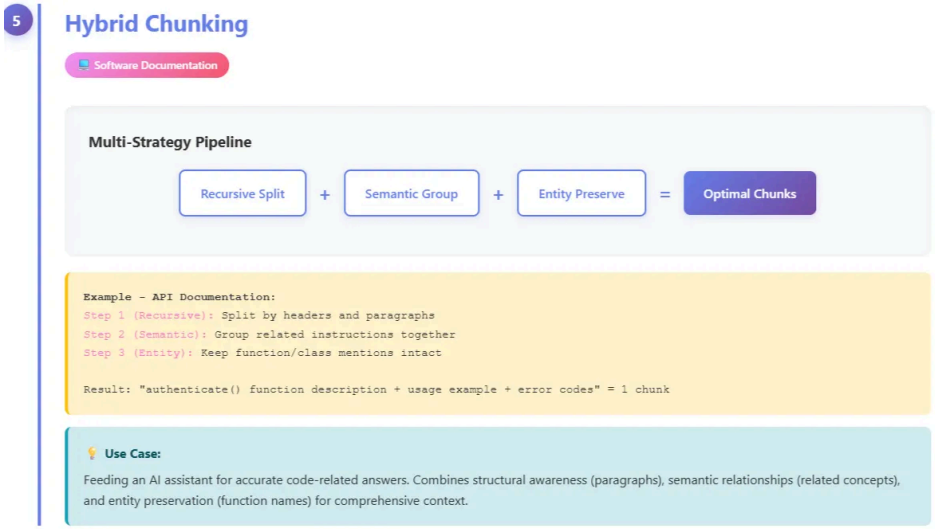
```
Step 1 - Recursive Split:
  Split by headers (##) and paragraph breaks (\n\n)

  → Section: "Authentication Methods"
  → Section: "Error Handling"
  → Section: "Rate Limiting"

Step 2 - Semantic Grouping:
  Group related instructions within each section

  → "authenticate() function" + "usage examples"
  → "error codes" + "retry logic"

Step 3 - Entity Preservation:
  Ensure function/class names aren't split

  → Keep "authenticate(username, password)" together
  → Keep "RateLimitError" class definition complete

Final Chunk Example:
  "The authenticate() function accepts two parameters...
   Usage: authenticate('user', 'pass')
   Returns: JWT token or raises AuthenticationError
   Related: See also refresh_token() method"
```

```
Hybrid Pipeline:

Raw Doc → Recursive Split → Semantic Group → Entity Preserve
           (Structure)      (Meaning)         (Completeness)
                                                    ↓
                                              ┌──────────┐
                                              │ Optimal Chunk │
                                              │ • Structured  │
                                              │ • Coherent    │
```

```
└──────────────┘
```

Use Case: Feeding an AI assistant for accurate code-related answers. Combines structural awareness (paragraphs), semantic relationships (related concepts), and entity preservation (function names) for comprehensive context. Results in chunks that are both technically accurate and semantically meaningful.

Implementation Tip: Order matters! Usually: structure → semantics → entities. Test different combinations for your domain.



6. Task-Aware Chunking

Domain: Code Analysis and QA Systems

Concept: Task-Aware Chunking adapts the chunking strategy based on the downstream task the LLM will perform.

How It Works:

- Define the target task (summarization, search, Q&A)

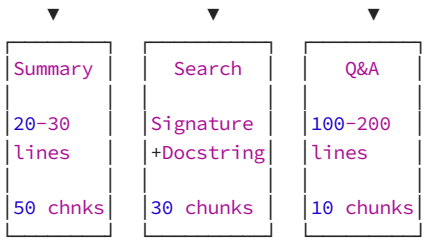
- Apply task-specific chunking rules
- Same content, different chunks for different tasks

Example:

Python codebase with 1000 lines:

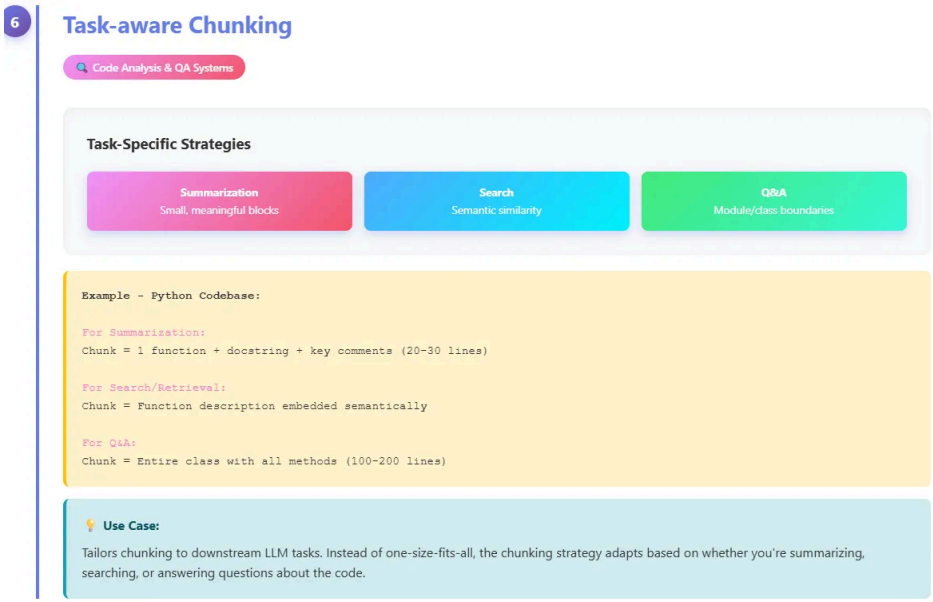
```
TASK: Summarization
Strategy: Small, meaningful blocks
Chunk Size: 20-30 lines per function

  Chunk 1:
    def authenticate(user, password):
        """Validates user credentials"""
        # Implementation...
        return token

TASK: Search/Retrieval
Strategy: Semantic similarity of descriptions
Chunk Size: Function signature + docstring

  Chunk 1:
    def authenticate(user, password):
        """Validates user credentials and returns JWT token"""

TASK: Question Answering
Strategy: Module/class boundaries
Chunk Size: 100-200 lines (entire class)

  Chunk 1:
    class AuthenticationManager:
        def __init__(self): ...
        def authenticate(self, user, password): ...
        def refresh_token(self, token): ...
        def logout(self, user): ...
```

```
Same Codebase, Different Tasks:

        ┌──────────────────┐
        │   Python Module  │
        │    (1000 lines)  │
        └──────────────────┘
                 │
        ┌────────┴─────────┐
        │                  │
```



Use Case: Tailors chunking to downstream LLM tasks. Instead of one-size-fits-all, the chunking strategy adapts based on whether you're summarizing, searching, or answering questions about the code. Significantly improves task-specific performance metrics.

Implementation Tip: Create a chunking configuration file with task-specific parameters. Consider using different strategies for the same corpus indexed for multiple purposes.



7. HTML/XML Tag-Based Splitting

Domain: Web Content and Scraped Documents

Concept: Tag-Based Splitting uses HTML/XML structure to determine chunk boundaries, respecting the document's hierarchical organization.

How It Works:

- Parse HTML/XML into DOM tree
- Define splitting tags (e.g., `<h2>`, `<section>`, `<div>`)
- Extract content within each structural boundary

Example:

Blog post HTML:

```
<article>
  <h1>Complete Guide to Machine Learning</h1>

  <h2>Introduction</h2>              ← CHUNK BOUNDARY
  <p>Machine learning has revolutionized...</p>
  <p>This guide covers fundamental concepts...</p>

  <h2>Supervised Learning</h2>       ← CHUNK BOUNDARY
  <p>Supervised learning algorithms...</p>
  <div class="example">
    <h3>Example: Linear Regression</h3>
    <p>Linear regression predicts...</p>
  </div>

  <h2>Unsupervised Learning</h2>     ← CHUNK BOUNDARY
  <p>Unlike supervised learning...</p>
  <ul>
    <li>Clustering algorithms</li>
    <li>Dimensionality reduction</li>
  </ul>
</article>

Resulting Chunks:

Chunk 1: Introduction section
  "Machine learning has revolutionized...
   This guide covers fundamental concepts..."

Chunk 2: Supervised Learning section
  "Supervised learning algorithms...
   Example: Linear Regression
```

```
    Linear regression predicts..."

Chunk 3: Unsupervised Learning section
  "Unlike supervised learning...
    • Clustering algorithms
    • Dimensionality reduction"
```

```
HTML Structure → Tag-Based Splitting:

<article>
  ├─ <h1> (ignored as title)
  │
  ├─ <h2>Introduction</h2>  ──────┐
  │   └─ <p>...</p>               ├──→ Chunk 1
  │   └─ <p>...</p>  ─────────────┘
  │
  ├─ <h2>Supervised</h2>  ────────┐
  │   └─ <p>...</p>               │
  │   └─ <div>                    ├──→ Chunk 2
  │       └─ <h3>...</h3>         │
  │       └─ <p>...</p>  ─────────┘
  │
  └─ <h2>Unsupervised</h2>  ──────┐
      └─ <p>...</p>               ├──→ Chunk 3
      └─ <ul>...</ul>  ───────────┘
```

Use Case: Perfect for feeding structured web content into an LLM for summarization, question answering, or content extraction. Preserves the logical structure of web pages by respecting HTML hierarchy, ensuring sections remain intact and contextually complete.

Implementation Tip: Use BeautifulSoup or lxml for parsing. Consider nested tags — should `<h3>` create sub-chunks or stay with `<h2>` parent

8. Code-Specific Splitting

Domain: Software Development

Concept: Code-Specific Splitting uses Abstract Syntax Tree (AST) parsing or language-specific patterns to split code at logical boundaries like functions, classes, or modules.

How It Works:

- Parse code into AST (Abstract Syntax Tree)
- Identify code units (classes, functions, methods)
- Create chunks at these natural boundaries

Example:

Python file with 500 lines:

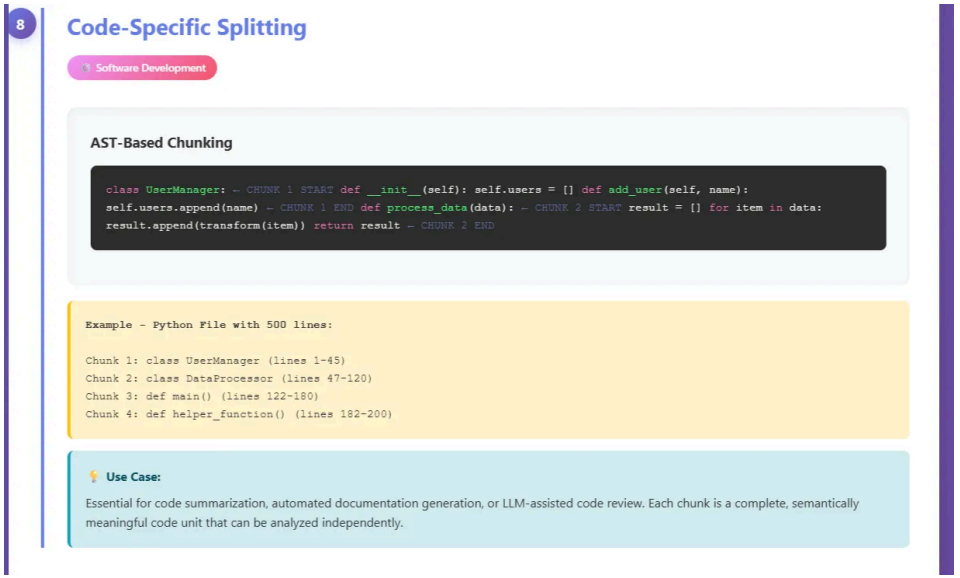
```
# Original File Structure:

class UserManager:                    ← CHUNK 1 START
```

```
    """Manages user operations"""

    def __init__(self):
        self.users = []

    def add_user(self, name, email):
        """Adds a new user"""
        user = User(name, email)
        self.users.append(user)
        return user

    def remove_user(self, user_id):
        """Removes user by ID"""
        self.users = [u for u in self.users
                      if u.id != user_id]
                                    ← CHUNK 1 END (lines 1-45)

class DataProcessor:                ← CHUNK 2 START
    """Processes data pipelines"""

    def __init__(self, config):
        self.config = config

    def process(self, data):
        """Main processing method"""
        cleaned = self.clean_data(data)
        transformed = self.transform(cleaned)
        return transformed
                                    ← CHUNK 2 END (lines 47-120)

def main():                         ← CHUNK 3 START
    """Main entry point"""
    manager = UserManager()
    processor = DataProcessor(config)
    # ... implementation
                                    ← CHUNK 3 END (lines 122-180)

def helper_function(x, y):          ← CHUNK 4 START
    """Utility function"""
    return x + y
                                    ← CHUNK 4 END (lines 182-200)
```

```
AST-Based Code Chunking:

Python File (500 lines)
        │
        ├──→ AST Parser
        │
        └──→ Code Units Detected:
             │
             ├─ class UserManager (lines 1-45)    ──→ Chunk 1
             │   ├─ __init__
             │   ├─ add_user
             │   └─ remove_user
             │
             ├─ class DataProcessor (lines 47-120) ──→ Chunk 2
             │   ├─ __init__
             │   └─ process
             │
             ├─ def main() (lines 122-180)         ──→ Chunk 3
             │
             └─ def helper_function() (182-200)    ──→ Chunk 4
```


Use Case: Essential for code summarization, automated documentation generation, or LLM-assisted code review. Each chunk is a complete, semantically meaningful code unit that can be analyzed independently. Prevents splitting classes or functions mid-implementation, which would lose critical context.

Implementation Tip: Use language-specific parsers: `ast` module for Python, Tree-sitter for multi-language support, regex for simple cases.



9. Regular Expression (Regex) Splitting

Domain: Log Analysis

Concept: Regex Splitting uses pattern matching to identify chunk boundaries based on predictable text patterns like timestamps, delimiters, or markers.

How It Works:

- Define a regex pattern for boundaries
- Split text whenever pattern is matched
- Each chunk starts with the matched pattern

Example:

Server log file with 10,000 entries:

```
Original Log File:

[2025-01-15 10:23:45] INFO: User john@example.com logged in successfully
[2025-01-15 10:24:12] ERROR: Database connection failed - timeout after 30s
[2025-01-15 10:24:15] INFO: Retrying database connection...
[2025-01-15 10:24:18] INFO: Connection restored successfully
[2025-01-15 10:25:30] WARN: High memory usage detected - 87% utilized
[2025-01-15 10:26:45] ERROR: API rate limit exceeded for endpoint /api/users

Regex Pattern: \[\d{4}-\d{2}-\d{2} \d{2}:\d{2}:\d{2}\]

Resulting Chunks:

Chunk 1:
  [2025-01-15 10:23:45] INFO: User john@example.com logged in successfully

Chunk 2:
  [2025-01-15 10:24:12] ERROR: Database connection failed - timeout after 30s

Chunk 3:
  [2025-01-15 10:24:15] INFO: Retrying database connection...

Chunk 4:
  [2025-01-15 10:24:18] INFO: Connection restored successfully

Chunk 5:
  [2025-01-15 10:25:30] WARN: High memory usage detected - 87% utilized

Chunk 6:
  [2025-01-15 10:26:45] ERROR: API rate limit exceeded for endpoint /api/users
```



```
Regex Pattern Matching:

Log Stream:

┌──────────────────────────────────────────────┐
│ [2025-01-15 10:23:45] INFO: User login...    │ ← Match!
│ [2025-01-15 10:24:12] ERROR: DB connection...│ ← Match!
│ [2025-01-15 10:24:15] INFO: Retrying...      │ ← Match!
│ [2025-01-15 10:24:18] INFO: Connection OK...  │ ← Match!
└──────────────────────────────────────────────┘
                    │
                    ├──→ Split on pattern:
                    │    \[\d{4}-\d{2}-\d{2} \d{2}:\d{2}:\d{2}\]
                    │
                    └──→ Individual Chunks:
                         Each log entry = 1 chunk

Advanced: Group by time windows (5-minute buckets):

┌──────────────────────────────────────────────┐
│ 10:23:00 - 10:28:00 (5 entries)              │ ← Chunk 1
├──────────────────────────────────────────────┤
│ 10:28:00 - 10:33:00 (12 entries)             │ ← Chunk 2
└──────────────────────────────────────────────┘
```

Use Case: Feeding structured log chunks into an LLM for anomaly detection, troubleshooting, or log summarization. Each entry is self-contained with its timestamp and context, making it easy to track issues chronologically. Can be enhanced with grouping by time windows or severity levels.

Implementation Tip: Common patterns:

- Timestamps: `\[\d{4}-\d{2}-\d{2}.*?\]` or `\d{4}/\d{2}/\d{2}`
- Section markers: `^={3,}` or `^-{3,}`
- Custom delimiters: `^###` or `^\*\*\*`

9

Regular Expression (Regex) Splitting

Log Analysis

Pattern-Based Splitting

```
[2025-01-15 10:23:45] INFO: User login successful ← CHUNK 1 [2025-01-15 10:24:12] ERROR: Database connection
failed ← CHUNK 2 [2025-01-15 10:24:15] INFO: Retrying connection... ← CHUNK 3 [2025-01-15 10:24:18] INFO:
Connection restored ← CHUNK 4 Regex pattern: \[\d{4}-\d{2}-\d{2} \d{2}:\d{2}:\d{2}\]
```

```
Example - Server Logs (10,000 lines):

Regex: \[\d{4}-\d{2}-\d{2} \d{2}:\d{2}:\d{2}\]

Split into 10,000 individual log entries
Each chunk = timestamp + log level + message
Can be further grouped by ERROR level or time windows
```

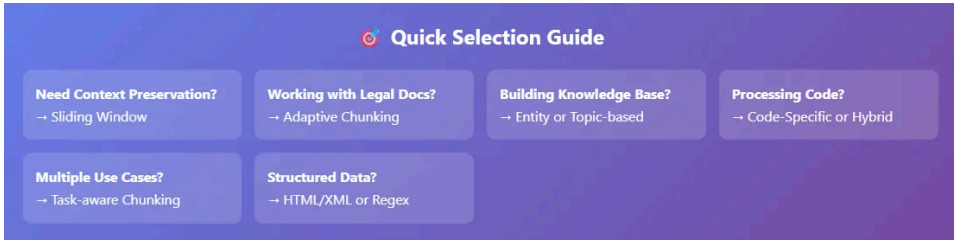
Use Case:

Feeding structured log chunks into an LLM for anomaly detection, troubleshooting, or log summarization. Each entry is self-contained with its timestamp and context, making it easy to track issues chronologically.

Quick Selection Guide

Choose based on your requirements:

- **Need context preservation across boundaries?** → Sliding Window
- **Working with legal documents?** → Adaptive Chunking
- **Building a knowledge base?** → Entity-based or Topic-based
- **Processing source code?** → Code-Specific or Hybrid
- **Multiple use cases for same content?** → Task-aware Chunking
- **Structured data (HTML/logs)?** → HTML/XML or Regex
- **Complex technical documentation?** → Hybrid (Recursive + Semantic + Entity)



Implementation Best Practices

1. **Start Simple:** Begin with basic strategies, add complexity as needed
2. **Measure Performance:** Track retrieval accuracy, response quality metrics
3. **Consider Costs:** Complex strategies (entity, topic) require more compute
4. **Test Chunk Sizes:** Experiment with different sizes for your LLM's context window
5. **Validate Boundaries:** Manually review sample chunks to ensure quality
6. **Version Control:** Track chunking strategy changes and their impact
7. **Monitor Drift:** Content types change; re-evaluate periodically

Final Thoughts!

Advanced chunking strategies go beyond simple text splitting to create context-aware, task-optimized, and structurally sound chunks. By understanding these 9 techniques and their appropriate use cases, you can significantly improve your LLM application's performance, retrieval accuracy, and user experience.

The key is matching the chunking strategy to your specific domain, content type, and downstream tasks. Don't hesitate to combine multiple approaches (hybrid chunking) or create custom strategies tailored to your unique requirements.

Artificial Intelligence | Retrieval Augmented Gen