Markdown Renderer: Design Document

Overview

A Markdown to HTML converter that transforms markdown text into semantic HTML through a multi-stage parsing pipeline. The key architectural challenge is handling the recursive nature of nested elements while maintaining proper precedence between block-level and inline elements.

> This guide is meant to help you understand the big picture before diving into each milestone. Refer back to it whenever you need context on how components connect.

Context and Problem Statement

> **Milestone(s):** Milestone 1: Block Elements, Milestone 2: Inline Elements, Milestone 3: Lists, Milestone 4: HTML Generation

At its core, building a markdown renderer means solving a **document transformation problem** — converting human-friendly lightweight markup into structured HTML that browsers can render. This challenge sits at the intersection of text processing, formal language parsing, and document structure understanding. The fundamental difficulty lies in the fact that markdown's informal, human-readable syntax must be transformed into HTML's strict, nested tag structure while preserving semantic meaning and handling edge cases gracefully.

The markdown parsing problem exemplifies what makes text processing both fascinating and challenging for software engineers. Unlike parsing structured data formats like JSON or XML, markdown parsing must handle ambiguous syntax, context-dependent interpretation, and the inherent messiness of human-authored content. A single line of text might be a heading, the start of a paragraph, part of a code block, or a list item — the parser must examine surrounding context, indentation patterns, and delimiter sequences to make the correct determination.

This design document addresses the architectural decisions needed to build a robust markdown renderer that can handle the full spectrum of CommonMark syntax while remaining maintainable and extensible. We focus on the critical parsing pipeline design, the intermediate representations that enable correct transformation, and the error handling strategies that ensure graceful degradation when encountering malformed input.

The Document Transformation Mental Model

Think of markdown parsing like **translating between two human languages with fundamentally different grammatical structures**. Imagine you're translating from a language that uses word order and context to convey meaning (like English) into a language that uses explicit grammatical particles and nested structures (like Japanese or Latin). The translator doesn't just substitute words — they must understand the semantic intent of entire phrases and restructure them according to the target language's rules.

In markdown parsing, the "source language" is markdown's lightweight, context-dependent syntax where meaning emerges from patterns like indentation, blank lines, and special character sequences. The "target language" is HTML's explicit, hierarchically nested tag structure where every element has clear boundaries and relationships. Just as a human translator must understand context and implied meaning, a markdown parser must infer document structure from visual patterns and formatting conventions.

Consider this concrete example of the transformation challenge:

```
# Getting Started

Here's a simple example:

    def hello():
        print("world")

This shows basic **Python** syntax.
```

The human reader immediately recognizes this as a heading, followed by introductory text, then a code block, and finally a concluding paragraph with emphasis. However, the parser must systematically analyze each line to make these determinations. The heading is identified by the `#` prefix pattern. The blank lines signal paragraph boundaries. The consistent four-space indentation indicates a code block. The `**` delimiters mark inline emphasis within the final paragraph.

The parser must handle this analysis in a way that mirrors how humans process the document — first identifying the overall block structure (heading, paragraph, code block, paragraph), then processing inline formatting within each block. This two-phase approach reflects how we naturally read documents: we first perceive the layout and major sections, then focus on detailed formatting within each section.

The translation analogy extends to error handling as well. When a human translator encounters ambiguous or incorrect source text, they make reasonable assumptions about intent and produce the most sensible translation possible. Similarly, markdown parsers must handle malformed input gracefully — treating unmatched delimiters as literal text, continuing parsing after encountering invalid syntax, and producing reasonable HTML output even from imperfect markdown input.

This mental model of **progressive structural interpretation** guides our architectural decisions. Like a translator working with complex nested clauses, our parser must maintain context about its current position in the document structure while making local decisions about individual elements. It must handle both the forest (overall document organization) and the trees (individual formatting elements) in a coordinated fashion.

Existing Parsing Approaches

The markdown parsing landscape offers three primary architectural approaches, each representing different trade-offs between implementation complexity, parsing accuracy, and maintainability. Understanding these approaches provides crucial context for our design decisions and helps illuminate why certain architectural patterns emerge as best practices.

> **Decision: Two-Phase Parsing Architecture**
>
> - **Context**: Markdown syntax exhibits both block-level structure (paragraphs, headings, lists) and inline formatting (emphasis, links, code spans). These two levels interact in complex ways that require different parsing strategies.
> - **Options Considered**: Regex-only approach, recursive descent parser, two-phase block-then-inline parser
> - **Decision**: Implement a two-phase parser that handles block structure first, then processes inline elements within each block
> - **Rationale**: Block structure provides the fundamental document skeleton and is largely context-independent, while inline parsing requires understanding of the containing block type. Separating these concerns simplifies both parsing phases and aligns with how humans read documents.
> - **Consequences**: Enables simpler, more maintainable parsing logic at the cost of requiring two complete passes through the document content.

Approach	Implementation Complexity	Parsing Accuracy	Performance	Maintainability	Error Recovery
Regex-Only	Low	Poor	High	Poor	Poor
Recursive Descent	High	High	Medium	Medium	Good
Two-Phase (Block + Inline)	Medium	High	Medium	Good	Good

The Regex-Only Approach represents the most straightforward implementation strategy, where each markdown construct maps to a regular expression pattern. A heading parser might use `^(#{1,6})\s+(.+)$` to match ATX-style headings, while emphasis detection could employ `\*\*([^*]+)\*\*` for bold text. This approach feels natural to developers familiar with text processing and offers immediate gratification — you can have basic markdown rendering working in under 100 lines of code.

However, the regex-only approach quickly encounters fundamental limitations that stem from markdown's context-sensitive nature. Consider the challenge of parsing emphasis correctly: the pattern `_underscore_emphasis_` should render as `<em>underscore_emphasis</em>` because underscores in the middle of words don't trigger emphasis according to CommonMark rules. A simple regex cannot encode this contextual understanding without becoming prohibitively complex. Additionally, nested structures like lists containing code blocks or blockquotes containing emphasis require sophisticated lookahead and backtracking that pushes regular expressions beyond their effective limits.

The error recovery characteristics of regex-based parsing prove particularly problematic. When a regular expression fails to match, the parser has little context about why the failure occurred or how to proceed. This leads to either silent failures (where malformed markdown disappears from output) or catastrophic failures (where a single syntax error breaks the entire parsing process). Real-world markdown documents contain numerous edge cases and minor syntax errors that require graceful handling.

The Recursive Descent Approach treats markdown as a formal grammar and builds a traditional recursive descent parser with productions for each syntactic construct. This approach offers excellent theoretical foundations and can handle arbitrarily complex nested structures with proper precedence handling. Parser generators or hand-written recursive functions naturally express the hierarchical relationships between markdown elements.

Recursive descent parsers excel at error recovery because they maintain rich context about the current parsing position and can implement sophisticated backtracking strategies. When parsing fails at one level, the parser can unwind to a previous state and attempt alternative interpretations. This leads to robust handling of malformed input and better diagnostic error messages.

The primary challenge with recursive descent parsing lies in markdown's inherent ambiguity and lookahead requirements. Unlike programming languages with unambiguous grammars, markdown often requires examining multiple lines ahead to make parsing decisions. Setext-style headings (where the heading text is followed by a line of `===` or `---` characters) exemplify this challenge — the parser cannot definitively identify a heading until it reads the following line. This lookahead requirement complicates the recursive descent approach and can lead to significant backtracking overhead.

The Two-Phase Parsing Approach emerged from the CommonMark specification's recognition that block-level and inline parsing represent fundamentally different challenges requiring different strategies. This approach first processes the entire document to identify block-level structures (paragraphs, headings, lists, code blocks, blockquotes), building an intermediate representation that captures the document's hierarchical organization. The second phase processes inline elements within each block, handling emphasis, links, images, and code spans according to rules specific to each block type.

The key insight driving two-phase parsing is that **block structure provides semantic context for inline parsing**. Code blocks disable all inline processing, preserving literal text exactly as written. List items require special handling of line breaks and

continuation. Headings process inline elements but ignore certain constructs like line breaks. By establishing block structure first, the inline parser operates with clear context about how to interpret formatting within each container.

Two-phase parsing also enables superior error handling through **graceful degradation**. If block parsing encounters malformed syntax, it can fall back to treating questionable content as paragraph text, allowing inline parsing to proceed normally. If inline parsing fails to match emphasis delimiters correctly, it can render them as literal text without affecting the overall document structure. This resilience mirrors how humans read documents — we can extract meaning even from imperfectly formatted text.

The intermediate representation between parsing phases provides a natural extension point for custom functionality. Additional block types, custom renderers, and syntax validation can all hook into the AST structure without modifying the core parsing logic. This architectural flexibility proves crucial for long-term maintainability and feature enhancement.

Parsing Phase	Input Format	Processing Strategy	Output Format	Error Handling
Block Parsing	Raw markdown text	Line-by-line state machine	Block-structured AST	Fallback to paragraph blocks
Inline Parsing	Text content from AST blocks	Delimiter-based pattern matching	Fully populated AST	Render unmatched delimiters as literal text
HTML Generation	Complete AST structure	Tree traversal with node-specific renderers	Valid HTML output	Escape invalid content, continue processing

The two-phase approach does introduce some complexity overhead compared to simpler alternatives. The parser must maintain sophisticated state between phases, and certain edge cases require coordination between block and inline processing. However, this complexity remains localized within well-defined architectural boundaries, making it manageable compared to the sprawling complexity that emerges from trying to handle all parsing concerns simultaneously.

> **The critical architectural insight is that document structure emerges at different levels of granularity, and each level requires specialized parsing strategies that align with how humans naturally process written content.**

Understanding these parsing approaches provides the foundation for our detailed design decisions in subsequent sections. The two-phase architecture shapes everything from our data model design to our error handling strategies, and recognizing the trade-offs involved helps explain why certain implementation choices emerge as architectural necessities rather than arbitrary preferences.

⚠️ **Pitfall: Attempting Single-Pass Parsing** Many developers initially attempt to parse both block structure and inline formatting in a single pass through the markdown text. This approach seems more efficient and simpler to implement. However, it quickly leads to complex, brittle code that struggles with context-dependent parsing rules. For example, determining whether `*` characters represent list bullets, emphasis markers, or literal text requires understanding the surrounding block context. Single-pass parsers either make incorrect parsing decisions or require extensive backtracking that negates any performance benefits. The two-phase approach, while requiring two passes through the content, results in significantly cleaner, more maintainable code that handles edge cases correctly.

Implementation Guidance

Our implementation strategy focuses on building a robust foundation that can grow with your understanding while providing immediate feedback on progress. The technology choices prioritize clarity and debuggability over performance optimization, recognizing that correct implementation comes before efficient implementation.

Technology Recommendations:

Component	Simple Option	Advanced Option	Rationale
Text Processing	String manipulation with `str.split()` and `str.strip()`	Regular expressions with `re` module	Start simple, add regex patterns as needed
Data Structures	Dictionaries and lists for AST nodes	Custom classes with type hints	Dictionaries are easier to debug and inspect
Input/Output	File reading with `open()` and string returns	Streaming with generators for large files	File-based I/O suffices for learning projects
Testing	`assert` statements with sample inputs	`unittest` or `pytest` framework	Simple assertions provide immediate feedback
HTML Generation	String concatenation and `.format()`	Template engines like `jinja2`	String operations keep dependencies minimal

Recommended File Structure:

```
markdown-renderer/
├── src/
│   ├── __init__.py
│   ├── lexer.py              ← Text tokenization utilities
│   ├── block_parser.py       ← Block-level element parsing
│   ├── inline_parser.py      ← Inline formatting parsing
│   ├── list_parser.py        ← List-specific parsing logic
│   ├── html_generator.py     ← AST to HTML conversion
│   └── ast_nodes.py          ← Data structures for parsed content
├── tests/
│   ├── __init__.py
│   ├── test_block_parser.py  ← Block parsing verification
│   ├── test_inline_parser.py ← Inline parsing verification
│   ├── test_integration.py   ← End-to-end parsing tests
│   └── fixtures/             ← Sample markdown files
│       ├── basic.md
│       ├── complex.md
│       └── edge_cases.md
├── examples/
│   ├── simple_example.py     ← Basic usage demonstration
│   └── sample_documents/     ← Test markdown files
└── main.py                   ← Command-line interface
```

This structure separates concerns clearly while keeping the project manageable. Each parsing phase gets its own module, making it easy to test components independently and understand their responsibilities. The `ast_nodes.py` module provides shared data structures, while `lexer.py` contains utilities used across multiple parsing phases.

Infrastructure Starter Code:

The following complete utility functions handle common text processing tasks that aren't central to learning parsing concepts but are necessary for implementation:

PYTHON

```python
# src/lexer.py - Complete utility functions

def normalize_line_endings(text):

    """Convert all line endings to Unix-style \n characters."""

    return text.replace('\r\n', '\n').replace('\r', '\n')


def split_into_lines(text):

    """Split text into lines, preserving information about blank lines."""

    lines = normalize_line_endings(text).split('\n')

    return [(i, line) for i, line in enumerate(lines)]


def is_blank_line(line):

    """Check if a line contains only whitespace characters."""

    return len(line.strip()) == 0


def get_indentation_level(line):

    """Calculate the number of leading spaces in a line."""

    return len(line) - len(line.lstrip(' '))


def remove_leading_spaces(line, num_spaces):

    """Remove up to num_spaces leading spaces from line."""

    spaces_to_remove = min(num_spaces, get_indentation_level(line))

    return line[spaces_to_remove:]


# src/html_generator.py - HTML escaping utilities

HTML_ESCAPE_TABLE = {

    '&': '&amp;',

    '<': '&lt;',

    '>': '&gt;',

    '"': '&quot;',

    "'": '&#x27;'

}


def escape_html(text):

    """Escape special HTML characters to prevent injection."""
```

```
    result = text
    for char, escape in HTML_ESCAPE_TABLE.items():
        result = result.replace(char, escape)
    return result

def pretty_print_html(html, indent_size=2):
    """Add indentation to HTML for readable output."""
    lines = []
    indent_level = 0

    for line in html.split('\n'):
        line = line.strip()
        if not line:
            continue

        if line.startswith('</'):
            indent_level -= 1

        lines.append(' ' * (indent_level * indent_size) + line)

        if line.startswith('<') and not line.startswith('</') and not line.endswith('/>'):
            indent_level += 1

    return '\n'.join(lines)
```

Core Parsing Infrastructure Skeleton:

The main parsing coordinator provides the overall structure while leaving the core logic for you to implement:

PYTHON

```python
# src/markdown_parser.py - Main parsing coordinator

class MarkdownParser:

    """Coordinates the two-phase parsing process from markdown to HTML."""



    def __init__(self):

        self.block_parser = BlockParser()

        self.inline_parser = InlineParser()

        self.html_generator = HTMLGenerator()



    def parse_to_html(self, markdown_text):

        """

        Convert markdown text to HTML through two-phase parsing.



        Args:

            markdown_text (str): Raw markdown content



        Returns:

            str: Valid HTML output

        """

        # TODO 1: Use lexer utilities to split text into lines and normalize formatting

        # TODO 2: Pass lines to block parser to build initial AST structure

        # TODO 3: Pass block AST to inline parser to process formatting within blocks

        # TODO 4: Pass complete AST to HTML generator for final output

        # TODO 5: Return escaped, properly formatted HTML

        pass



    def parse_file(self, file_path):

        """Parse a markdown file and return HTML output."""

        # TODO 1: Open file and read contents with proper encoding handling

        # TODO 2: Call parse_to_html with file contents

        # TODO 3: Handle file reading errors gracefully
```

```
        pass

# src/ast_nodes.py - Data structure skeletons
class ASTNode:
    """Base class for all AST nodes in the parsing tree."""

    def __init__(self, node_type):
        # TODO: Define common fields needed by all AST nodes
        # Hint: node_type, children, parent, line_number are useful
        pass

class BlockNode(ASTNode):
    """Base class for block-level elements like paragraphs and headings."""

    def __init__(self, block_type):
        # TODO: Initialize base class and add block-specific fields
        # Hint: Block nodes contain inline content and have block-specific attributes
        pass

class InlineNode(ASTNode):
    """Base class for inline elements like emphasis and links."""

    def __init__(self, inline_type):
        # TODO: Initialize base class and add inline-specific fields
        # Hint: Inline nodes often have text content and formatting attributes
        pass
```

Language-Specific Implementation Hints:

Python's string processing capabilities make it an excellent choice for text parsing projects. Key language features that simplify markdown parsing include:

- **String slicing**: Use `line[2:]` to remove leading `## ` from headings, `text[start:end]` for extracting content between delimiters
- **String methods**: `str.startswith()` for prefix detection, `str.strip()` for whitespace removal, `str.count()` for delimiter counting

- **List comprehensions**: `[line for line in lines if not is_blank_line(line)]` for filtering, `[get_indentation_level(line) for line in block]` for indentation analysis
- **Regular expressions**: Import `re` module for complex pattern matching, use `re.compile()` to precompile frequently used patterns for better performance
- **Dictionary-based state machines**: Use dictionaries to map states to handler functions, enabling clean state machine implementation without complex if-elif chains

When implementing the parsing logic, prefer explicit state tracking over implicit assumptions. Maintain variables like `current_block_type`, `indentation_stack`, and `delimiter_stack` to make parsing decisions transparent and debuggable. Use descriptive variable names like `heading_level` instead of `level`, `emphasis_delimiter` instead of `delim` — the extra verbosity pays dividends when debugging complex parsing edge cases.

Debugging and Verification Strategies:

Since markdown parsing involves complex text transformations, robust debugging capabilities are essential. Build debugging support into your parser from the beginning:

```python
# Debug utilities to include in your implementation

def debug_print_ast(node, indent=0):

    """Print AST structure with indentation showing hierarchy."""

    # TODO: Implement tree traversal that shows node types and content

    # This helps verify that parsing creates the expected structure

    pass


def debug_print_parsing_steps(parser, text):

    """Show step-by-step parsing decisions for debugging."""

    # TODO: Add logging to show which parsing rules match each line

    # This helps identify where parsing goes wrong on malformed input

    pass


def validate_html_output(html):

    """Basic validation that generated HTML is well-formed."""

    # TODO: Check that all opening tags have matching closing tags

    # TODO: Verify that special characters are properly escaped

    # This catches common HTML generation errors before manual testing

    pass
```

The key to successful markdown parser implementation is building incrementally with constant verification. After implementing each parsing component, test it thoroughly with both valid and malformed input before moving to the next component. The two-phase architecture makes this incremental approach natural — you can verify block parsing completely before touching inline parsing, then verify inline parsing before implementing HTML generation.

Goals and Non-Goals

> **Milestone(s):** Milestone 1: Block Elements, Milestone 2: Inline Elements, Milestone 3: Lists, Milestone 4: HTML Generation

The Scope Definition Mental Model

Think of building a markdown renderer like planning a house construction project. Just as a house contractor must clearly define what's included in the base contract versus what constitutes expensive custom work, we must establish precise boundaries for our markdown renderer. Without clear scope boundaries, feature creep will transform a beginner-friendly learning project into an overwhelming enterprise-grade parsing system. The goals and non-goals serve as our project contract, protecting both the implementation timeline and the learning experience.

Every markdown renderer exists within a spectrum of complexity. On one end lie minimal parsers that handle basic formatting with simple regular expressions. On the other end sit full-featured systems like GitHub's markdown processor that support dozens of extensions, custom syntax highlighting, mathematical notation, and complex table formatting. Our markdown renderer intentionally positions itself as a comprehensive but focused implementation that teaches core parsing principles without drowning learners in edge cases and rarely-used features.

The key insight is that markdown parsing involves two distinct types of complexity: **structural complexity** (how we parse and represent markdown) and **feature complexity** (what markdown syntax we support). Our goals maximize structural complexity to teach essential parsing concepts while carefully limiting feature complexity to maintain project feasibility. This approach ensures learners experience the full architectural challenges of building a document transformer without getting lost in specification minutiae.

Primary Goals

The markdown renderer has four primary objectives that directly align with the core learning outcomes for building document transformation systems.

Goal 1: Implement Core CommonMark Block Elements

Our renderer will support the fundamental block-level structures that form the backbone of structured documents. This includes ATX-style headings (hash-prefixed), paragraphs with proper line handling, fenced and indented code blocks, horizontal rules, and blockquotes with nesting support. These elements teach the essential concept of **block-level parsing**, where we must identify document structure by analyzing line patterns and maintaining parsing state across multiple lines.

The structural learning value comes from implementing the state machine logic needed to distinguish between different block types, handle multi-line content correctly, and manage the transition between parsing phases. Code blocks, for example, require different parsing rules than regular paragraphs—content inside code blocks must be preserved literally without further markdown processing.

Goal 2: Implement Essential Inline Formatting Elements

Within the content of block elements, our renderer will parse inline formatting including emphasis (bold and italic using asterisks and underscores), inline code spans with backtick delimiters, hyperlinks using bracket-parenthesis syntax, and images using the exclamation-bracket-parenthesis pattern. These elements introduce the critical challenge of **nested parsing**, where formatting elements can contain other formatting elements with proper precedence rules.

The key learning objective is mastering **delimiter matching**—the algorithm for finding pairs of formatting markers while handling mismatched delimiters, escaped characters, and nested structures correctly. This teaches pattern recognition, state

management, and the recursive nature of text processing.

Goal 3: Implement Hierarchical List Structures

Both ordered and unordered lists with full nesting support represent one of the most complex structural parsing challenges in markdown. Lists require tracking indentation levels, distinguishing between list continuation and new list items, handling mixed content within list items, and building proper tree structures for nested lists. This goal teaches **recursive parsing** and **indentation-based structure detection**.

Lists also introduce the concept of **context-dependent parsing**, where the meaning of a line depends on its position relative to other elements. A line starting with `1.` might begin an ordered list, continue an existing list, or simply be regular paragraph content, depending on indentation and surrounding context.

Goal 4: Generate Valid, Well-Formed HTML Output

The final parsing phase converts our internal AST representation into semantic HTML5. This includes proper HTML entity escaping for special characters, correct nesting of HTML elements, and optionally formatted output with consistent indentation. This goal teaches **tree traversal algorithms**, **character encoding safety**, and the principles of generating structured output from internal representations.

HTML generation also introduces the concept of **separation of concerns** between parsing and rendering. The same AST could theoretically generate different output formats (HTML, XML, plain text) with different renderer implementations.

Core Goal	Primary Learning Concept	Key Implementation Challenge	Success Metric
Block Elements	State machine parsing	Line-by-line content classification	Correctly parse mixed block types in sequence
Inline Formatting	Delimiter matching and nesting	Handling nested emphasis and mismatched delimiters	Bold text can contain italic text and vice versa
Hierarchical Lists	Recursive structure building	Indentation tracking and list continuation	Three-level nested lists render correctly
HTML Generation	Tree traversal and escaping	Safe character encoding and valid markup	Output passes W3C HTML5 validation

Secondary Goals

Beyond the core parsing functionality, our markdown renderer includes several secondary objectives that enhance the learning experience and prepare for real-world usage scenarios.

Comprehensive Error Handling and Recovery

Rather than failing on malformed input, our renderer implements **graceful degradation** strategies that produce reasonable output even from invalid markdown. When encountering mismatched emphasis delimiters like `**bold text*`, the renderer should treat the unmatched markers as literal text rather than crashing or producing invalid HTML. This teaches defensive programming and the importance of robust parsers in production systems.

Error recovery also includes handling edge cases like empty inputs, files with only whitespace, unclosed code blocks, and deeply nested structures. Each error scenario provides learning opportunities about input validation and parser resilience.

Extensible Architecture with Plugin Points

The renderer design includes well-defined interfaces that allow customization of HTML output without modifying core parsing logic. This might include custom CSS class injection, alternative HTML element choices, or specialized rendering for code blocks with syntax highlighting hooks. This secondary goal teaches **interface design** and **extensibility patterns** common in larger software systems.

The plugin architecture demonstrates how to build systems that support customization while maintaining a stable core. Even if learners don't implement custom renderers, understanding the extension points teaches valuable lessons about API design.

Performance-Conscious Implementation

While performance is not the primary objective, our implementation avoids obviously inefficient patterns that would make the renderer unusable on moderately-sized documents. This means single-pass parsing where possible, avoiding excessive string copying, and using appropriate data structures for different operations. This goal teaches **algorithmic efficiency** awareness without requiring advanced optimization techniques.

Performance consciousness also includes memory usage patterns—building AST nodes incrementally rather than storing entire document content in memory simultaneously, and cleaning up intermediate parsing state appropriately.

Explicit Non-Goals

Understanding what we will NOT implement is equally important for maintaining project scope and completion timeline. These non-goals represent features that would significantly increase implementation complexity without proportional learning value for a beginner-level project.

Advanced CommonMark Extensions

We will not implement tables, strikethrough text, task lists, footnotes, definition lists, or mathematical notation. While these features appear in many markdown flavors, they introduce substantial parsing complexity without teaching fundamentally new concepts. Tables alone require multi-pass parsing, column alignment algorithms, and complex HTML generation logic that could double the project timeline.

The decision to exclude extensions focuses learning energy on mastering the core parsing patterns rather than memorizing specification details for rarely-used syntax features.

Custom Syntax Extensions

Unlike some markdown processors, our renderer will not support custom syntax through configuration or plugins. Features like custom block types, alternative link syntax, or embed codes for external content fall outside our scope. Supporting custom syntax requires building a parser generator or macro system, which represents a completely different architectural challenge.

This boundary keeps the project focused on understanding markdown parsing specifically rather than general-purpose language processing.

Advanced HTML Features and Customization

We will not generate custom CSS classes, support HTML attributes in markdown syntax, or provide extensive HTML customization options. The HTML output will be clean, semantic HTML5 without styling concerns. While some markdown processors allow syntax like `{.css-class}` to add attributes, this feature adds parsing complexity without teaching core document transformation concepts.

Similarly, we won't support inline HTML pass-through beyond basic safety (HTML tags in markdown content should be escaped, not rendered). Supporting mixed HTML/markdown requires a much more sophisticated parser that can switch between parsing modes dynamically.

Performance Optimization and Streaming

Our implementation will not focus on streaming large documents, incremental parsing, or memory optimization for massive files. While production markdown processors often implement these features, they require advanced techniques like lazy evaluation, parser combinators, or custom memory management that would obscure the fundamental parsing algorithms we're trying to teach.

The renderer should handle documents up to several hundred kilobytes efficiently, which covers the vast majority of real-world markdown files without requiring optimization complexity.

Specification Compliance Edge Cases

We will not implement every edge case detail from the CommonMark specification, particularly around whitespace handling, Unicode normalization, or obscure delimiter precedence rules. For example, the exact behavior of emphasis markers inside words (`middle_of_word` should not trigger emphasis) is important for production systems but adds implementation complexity that doesn't teach new concepts.

Our approach prioritizes implementing the common usage patterns correctly rather than achieving 100% specification compliance on edge cases that rarely appear in real documents.

Non-Goal Category	Specific Excluded Features	Rationale	Alternative Learning Path
Advanced Extensions	Tables, footnotes, math notation	Doubles complexity without new parsing concepts	Separate advanced project after completing core renderer
Custom Syntax	Plugin-based syntax, custom blocks	Requires parser generator architecture	Study existing parser frameworks like ANTLR
HTML Customization	CSS injection, attribute syntax	HTML generation becomes dominant complexity	Separate CSS/styling project
Performance	Streaming, memory optimization	Obscures fundamental algorithms	Performance engineering course after mastering basics
Specification Compliance	Unicode edge cases, obscure precedence	Implementation detail memorization vs. concept learning	Read CommonMark spec for completeness understanding

Success Criteria and Acceptance Boundaries

To maintain clear project scope, we define specific success criteria that distinguish between acceptable implementation variations and scope creep that violates our goals.

Functional Success Criteria

A successful markdown renderer implementation must correctly parse and render a representative test document containing all supported elements in various combinations. This includes nested lists with inline formatting, code blocks adjacent to headings, blockquotes containing emphasis and links, and proper paragraph separation throughout.

The renderer must also handle empty inputs gracefully, process files with different line ending conventions (Unix, Windows, Mac), and generate HTML that validates against W3C standards without warnings or errors.

Quality Success Criteria

Beyond functional correctness, successful implementations demonstrate clean separation between parsing phases, readable code organization that follows the recommended file structure, and error handling that provides meaningful feedback rather than cryptic failure messages.

The AST structure should be inspectable for debugging, with utility functions that can pretty-print the parsed tree structure. This requirement ensures learners understand their parser's internal state, which is crucial for debugging and validation.

Learning Objective Success Criteria

Most importantly, successful implementations demonstrate that the learner understands the core concepts rather than just copying working code. This means they can explain why the two-phase parsing approach (block then inline) is necessary, how delimiter matching algorithms handle nested formatting, and what trade-offs their architectural decisions introduce.

Learners should be able to extend their implementation with simple new features (like supporting `~~strikethrough~~` syntax) without requiring major architectural changes. This demonstrates they've internalized the parsing patterns rather than just implementing specific features.

> **Key Design Principle**: Scope boundaries serve learning objectives, not feature completeness. Every included feature must teach a distinct parsing concept, while excluded features should be omittable without compromising the educational value of core implementation challenges.

The tension between scope and learning occurs when interesting features (like tables) would teach valuable lessons (like multi-pass parsing) but at the cost of project complexity that overwhelms beginners. Our goals resolve this tension by prioritizing depth in core concepts over breadth in supported features. Learners who master our focused implementation will be well-prepared to tackle extensions and advanced features in subsequent projects.

Implementation Guidance

The goals and non-goals translate into specific technology choices and project organization strategies that support successful implementation while maintaining educational focus.

A. Technology Recommendations Table:

Component	Simple Option	Advanced Option
Text Processing	Built-in string methods + basic regex	Full regex engines with lookahead/lookbehind
Data Structures	Lists and dictionaries for AST nodes	Custom tree classes with traversal methods
HTML Generation	String concatenation with escaping	Template engines or HTML builder libraries
File I/O	Read entire file into memory	Streaming/chunked file processing
Testing Framework	Built-in unittest/assert modules	Property-based testing frameworks
Code Organization	Single module with classes	Multi-package architecture with interfaces

For beginners, the simple options provide immediate productivity without requiring mastery of complex libraries. Advanced options become attractive once the core concepts are solid.

B. Recommended File/Module Structure:

The file organization should reflect the parsing pipeline and make the architectural boundaries clear:

```
markdown-renderer/
├── main.py                    # CLI entry point and file I/O
├── parser.py                  # MarkdownParser class and parse_to_html()
├── lexer.py                   # Line processing and normalization utilities
├── blocks.py                  # Block-level parsing (BlockNode classes)
├── inlines.py                 # Inline parsing (InlineNode classes)
├── lists.py                   # List parsing (separate due to complexity)
├── ast_nodes.py               # ASTNode, BlockNode, InlineNode definitions
├── html_generator.py          # HTML output generation and escaping
├── utils.py                   # Helper functions (debug_print_ast, etc.)
├── test_samples/              # Sample markdown files for testing
│   ├── basic_elements.md      # Simple test cases for each element
│   ├── nested_structures.md   # Complex combinations and nesting
│   └── edge_cases.md          # Error conditions and malformed input
└── tests/                     # Unit tests organized by component
    ├── test_blocks.py
    ├── test_inlines.py
    ├── test_lists.py
    └── test_integration.py
```

This structure separates concerns while keeping related functionality together. The `test_samples/` directory provides immediate feedback during development.

C. Infrastructure Starter Code (COMPLETE, ready to use):

PYTHON

```python
# utils.py - Complete utility functions for debugging and line processing

HTML_ESCAPE_TABLE = {
    '&': '&amp;',
    '<': '&lt;',
    '>': '&gt;',
    '"': '&quot;',
    "'": '&#39;'
}

def normalize_line_endings(text):
    """Convert all line endings to Unix format (\n)."""
    return text.replace('\r\n', '\n').replace('\r', '\n')

def split_into_lines(text):
    """Split text into lines, preserving empty lines and tracking line numbers."""
    lines = text.split('\n')
    return [(i + 1, line) for i, line in enumerate(lines)]

def is_blank_line(line):
    """Check if line contains only whitespace characters."""
    return len(line.strip()) == 0

def get_indentation_level(line):
    """Count leading spaces in line. Tabs count as 4 spaces."""
    count = 0
    for char in line:
        if char == ' ':
            count += 1
        elif char == '\t':
            count += 4
        else:
            break
```

```
    return count


def escape_html(text):
    """Escape special HTML characters to prevent injection."""
    if not text:
        return text
    result = text
    for char, escape in HTML_ESCAPE_TABLE.items():
        result = result.replace(char, escape)
    return result


def debug_print_ast(node, indent=0):
    """Print AST structure for debugging purposes."""
    prefix = "  " * indent
    if hasattr(node, 'node_type'):
        print(f"{prefix}{node.node_type}: {getattr(node, 'text_content', '')[:50]}")
    if hasattr(node, 'children'):
        for child in node.children:
            debug_print_ast(child, indent + 1)
```

D. Core Logic Skeleton Code (signature + TODOs only):

PYTHON

```python
# parser.py - Main parser class skeleton

class MarkdownParser:
    """Main markdown parser coordinating block and inline parsing phases."""

    def __init__(self):
        # TODO 1: Initialize block_parser, inline_parser, html_generator components
        # TODO 2: Set up shared state like current line number, parsing context
        pass

    def parse_to_html(self, markdown_text):
        """
        Convert markdown text to HTML through two-phase parsing.

        Args:
            markdown_text (str): Raw markdown input

        Returns:
            str: Generated HTML output
        """
        # TODO 1: Normalize line endings using normalize_line_endings()
        # TODO 2: Split into lines with split_into_lines()
        # TODO 3: Pass lines to block parser to build initial AST
        # TODO 4: Pass block AST to inline parser to process formatting
        # TODO 5: Pass complete AST to HTML generator for output
        # TODO 6: Return final HTML string
        pass

    def parse_file(self, file_path):
        """Parse markdown file and return HTML."""
        # TODO 1: Open and read file content
```

```
    # TODO 2: Call parse_to_html() with file content
    # TODO 3: Handle file I/O errors gracefully
    pass
```

E. Language-Specific Hints:

For Python implementation:

- Use `re.compile()` to pre-compile frequently used regex patterns for better performance
- List comprehensions are excellent for filtering and transforming line collections
- The `enumerate()` function simplifies line number tracking during parsing
- Consider using `dataclasses` for AST node definitions to reduce boilerplate
- The `textwrap.dedent()` function helps with processing indented code blocks
- Use `isinstance()` checks rather than string comparisons for node type detection

F. Milestone Checkpoint:

After establishing goals and non-goals, verify your project setup:

Setup Verification Steps:

1. Create the recommended file structure with empty Python files
2. Copy the complete utility functions into `utils.py`
3. Run `python -c "from utils import normalize_line_endings; print('Setup OK')"` to verify imports work
4. Create a simple test file `test_samples/basic.md` with a heading and paragraph
5. Verify you can read the test file and print its content

Expected Behavior:

- All imports should work without errors
- The utility functions should handle basic text processing correctly
- You should be able to process line endings and count indentation accurately
- File I/O should work for reading test markdown samples

Signs Something Is Wrong:

- Import errors suggest file structure problems or missing `__init__.py` files
- Utility functions failing indicates issues with the starter code setup
- File reading problems suggest path or permission issues

What to Check:

- Verify Python environment has required standard library modules
- Ensure all files are in the correct directory structure
- Check that test sample files use the line ending format your system expects
- Confirm utility functions produce expected output on sample inputs

This milestone checkpoint ensures learners have a solid foundation before beginning the actual parsing implementation in subsequent milestones.

High-Level Architecture

> **Milestone(s):** Milestone 1: Block Elements, Milestone 2: Inline Elements, Milestone 3: Lists, Milestone 4: HTML Generation

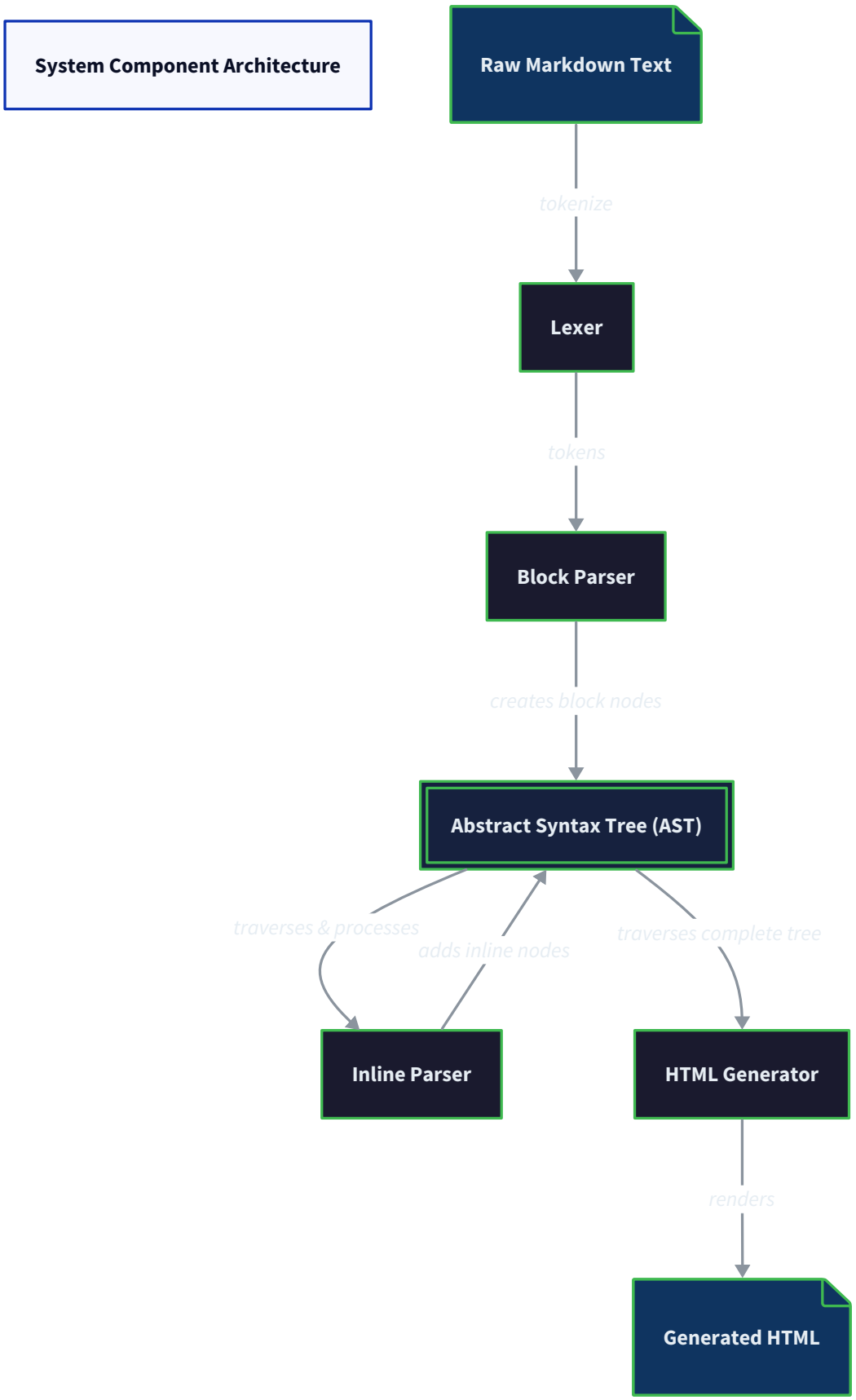
The Document Pipeline Mental Model

Think of markdown parsing like a factory assembly line for translating documents. Raw markdown text enters one end, and polished HTML emerges from the other. The key insight is that this transformation happens in distinct phases, each with specialized workers who understand different aspects of the document structure.

Imagine you're translating a complex technical document from one language to another. You wouldn't try to translate every word simultaneously while also formatting paragraphs and checking grammar. Instead, you'd first identify the major sections (chapters, headings, paragraphs), then work on the detailed sentence structure within each section, and finally apply the target language's formatting conventions. This is exactly how our **two-phase parsing** approach works.

The first phase identifies the document's skeleton - the major structural elements like headings, paragraphs, code blocks, and lists. These are **block-level elements** that define the document's overall organization. The second phase focuses on the details within each block - the **inline elements** like bold text, links, and inline code that provide formatting and semantic meaning within the content.

This separation is crucial because block and inline elements follow different parsing rules. A heading can contain italic text, but italic text cannot contain a heading. By parsing blocks first and then processing inline content within those blocks, we maintain proper precedence and avoid complex interdependencies.


System Component Architecture
Raw Markdown Text
tokenize
Lexer
tokens
Block Parser
creates block nodes
Abstract Syntax Tree (AST)
traverses & processes
adds inline nodes
traverses complete tree
Inline Parser
HTML Generator
renders
Generated HTML

Component Overview

The markdown renderer follows a **four-component architecture** where each component has a clearly defined responsibility in the document transformation pipeline. This separation of concerns makes the system easier to understand, test, and extend while ensuring that each component can focus on its specific parsing challenges.

Input Preprocessor and Lexer

The **Input Preprocessor** serves as the system's entry point, handling the messy realities of text input before any parsing begins. Think of it as a document sanitizer that ensures all components downstream can make consistent assumptions about the input format.

Responsibility	Description	Input	Output
Line Ending Normalization	Convert Windows CRLF and classic Mac CR to Unix LF	Raw markdown text	Normalized text
Line Splitting	Break text into individual lines while preserving line numbers	Normalized text	List of lines with metadata
Blank Line Detection	Identify whitespace-only lines for block boundary detection	Individual lines	Boolean classification
Indentation Analysis	Measure leading whitespace for list and code block processing	Individual lines	Indentation levels

The preprocessor doesn't make parsing decisions - it simply ensures that subsequent components receive clean, predictable input. This includes normalizing different line ending conventions, preserving empty lines that serve as block separators, and calculating indentation levels that will be crucial for list parsing.

> **Key Design Insight**: The preprocessor eliminates platform-specific text handling concerns from the core parsing logic. Without this component, every parser would need to handle Windows CRLF, Unix LF, and classic Mac CR line endings, significantly complicating the parsing state machines.

Block Parser

The **Block Parser** is responsible for identifying and parsing the document's structural skeleton. It processes the input line-by-line, maintaining state to recognize multi-line constructs and building the first level of the document's **Abstract Syntax Tree**.

Block Type	Recognition Pattern	State Requirements	Output Node Type
ATX Headings	Lines starting with 1-6 `#` characters	Single-line, immediate recognition	`BlockNode` with `block_type="heading"`
Paragraphs	Consecutive non-blank lines not matching other patterns	Multi-line accumulation state	`BlockNode` with `block_type="paragraph"`
Fenced Code Blocks	Lines between triple backticks	Start/end delimiter matching	`BlockNode` with `block_type="code_block"`
Indented Code Blocks	Lines indented by 4+ spaces	Consecutive indentation tracking	`BlockNode` with `block_type="code_block"`
Blockquotes	Lines starting with `>` character	Nested depth tracking	`BlockNode` with `block_type="blockquote"`
Horizontal Rules	Lines with 3+ dashes or asterisks	Pattern matching with spacing rules	`BlockNode` with `block_type="horizontal_rule"`

The block parser operates as a **state machine** where each line can potentially trigger a state transition. For example, when processing a paragraph, encountering a blank line transitions to a "seeking next block" state, while encountering an ATX heading immediately closes the paragraph and starts processing the heading.

> **Architecture Decision: Line-by-Line State Machine vs Regex Chunking**
>
> - **Context**: Block parsing requires handling multi-line constructs like paragraphs and code blocks
> - **Options Considered**:
>   1. Process entire input with complex regex patterns
>   2. Line-by-line state machine
>   3. Recursive descent parser
> - **Decision**: Line-by-line state machine
> - **Rationale**: Regex becomes unwieldy for nested constructs and lookahead requirements. Recursive descent adds complexity for what is fundamentally a sequential process. State machine provides clear separation of concerns and easier debugging.
> - **Consequences**: Enables incremental processing, simplifies testing of individual block types, but requires careful state transition management.

Inline Parser

The **Inline Parser** processes the text content within each block element, identifying and parsing formatting elements like emphasis, links, images, and inline code. This component operates on a fundamentally different principle than the block parser - it must handle **nested and overlapping constructs** within a single line or text span.

Inline Type	Delimiter Pattern	Nesting Rules	Special Handling
Strong (Bold)	`**text**` or `__text__`	Can contain emphasis	Cannot cross word boundaries for underscores
Emphasis (Italic)	`*text*` or `_text_`	Can be nested in strong	Cannot cross word boundaries for underscores
Inline Code	`code`	Cannot contain other formatting	Preserves literal content, including backticks
Links	`[text](url)`	Text portion can contain formatting	URL portion is literal
Images	`![alt](url)`	Alt text can contain formatting	URL portion is literal
Autolinks	`<url>` or `<email>`	No nested formatting	Automatic protocol detection

The inline parser uses a **delimiter matching algorithm** that maintains a stack of opening delimiters and matches them with closing delimiters according to CommonMark precedence rules. This is significantly more complex than block parsing because inline elements can be nested (bold text containing italic text) and can have complex interaction rules.

> **Key Design Insight**: Inline parsing cannot be solved with simple regex patterns because of the nesting requirements and context-dependent rules. For example, underscores in the middle of words should not trigger emphasis, and unmatched delimiters should be treated as literal text.

HTML Generator

The **HTML Generator** traverses the completed AST and produces valid HTML output. This component is responsible for **character escaping**, **proper tag nesting**, and **output formatting**. Think of it as the final assembly worker who takes the structured document representation and produces the final deliverable.

Generation Phase	Input	Process	Output
Tree Traversal	Complete AST	Depth-first traversal of nodes	Sequential processing order
Character Escaping	Text content	Convert `&`, `<`, `>`, `"` to HTML entities	Escaped text safe for HTML
Tag Generation	AST nodes with types	Map node types to HTML elements	Opening and closing tags
Attribute Handling	Node attributes (links, images, headings)	Generate HTML attributes	`href`, `src`, `alt`, `id` attributes
Pretty Printing	Generated HTML	Add indentation and line breaks	Human-readable formatted output

The generator must handle **self-closing tags** correctly (like `<hr>` for horizontal rules), ensure proper **tag nesting** (no unclosed or incorrectly ordered tags), and provide **extensibility** for custom renderers that might want to generate different output formats.

> **Architecture Decision: Early vs Late Character Escaping**
>
> - **Context**: HTML special characters must be escaped, but timing affects complexity
> - **Options Considered**:
>   1. Escape during initial input processing
>   2. Escape during inline parsing
>   3. Escape during HTML generation
> - **Decision**: Escape during HTML generation
> - **Rationale**: Early escaping interferes with parsing logic that needs to recognize literal characters. Late escaping ensures we only escape content that will be output as text, not structural markup.
> - **Consequences**: Cleaner parsing logic, but requires careful tracking of what content needs escaping vs what is already HTML.

Component Interaction and Data Flow

The components form a **pipeline architecture** where data flows unidirectionally from input to output, with each component transforming the data into a more structured representation.

Data Flow Stage	Data Format	Component	Key Transformations
Raw Input	String with mixed line endings	Input Preprocessor	Normalization and line splitting
Normalized Lines	List of strings with metadata	Block Parser	Structure identification and AST building
Block AST	Tree of `BlockNode` objects	Inline Parser	Text content parsing and inline AST integration
Complete AST	Mixed tree of `BlockNode` and `InlineNode` objects	HTML Generator	HTML element generation and formatting
HTML Output	Valid HTML string	External consumer	Ready for browser or file output

Each component maintains **clear interface boundaries** with well-defined input and output contracts. The block parser doesn't need to understand inline formatting, and the HTML generator doesn't need to understand markdown syntax. This separation enables independent testing, development, and potential replacement of individual components.

> **Critical Design Principle**: Each component operates on a higher level of abstraction than its predecessor. The preprocessor works with raw text, the block parser works with lines and structure, the inline parser works with text spans and formatting, and the HTML generator works with semantic document elements.

Recommended File Structure

The file organization follows the principle of **component isolation** with clear separation between parsing logic, data structures, and utilities. This structure supports incremental development where each milestone can be implemented and tested independently.

```
markdown_renderer/
├── __init__.py                 # Main package interface
├── main.py                     # CLI entry point and file processing
├── parser.py                   # MarkdownParser class and main API
├── data_model.py               # AST node definitions and data structures
├── preprocessor.py             # Input normalization and line processing
├── block_parser.py             # Block-level element parsing
├── inline_parser.py            # Inline formatting parsing
├── list_parser.py              # Specialized list parsing logic
├── html_generator.py           # AST to HTML conversion
├── utils.py                    # Shared utilities and constants
├── tests/                      # Test suite organization
│   ├── __init__.py
│   ├── test_preprocessor.py    # Input processing tests
│   ├── test_block_parser.py    # Block parsing tests
│   ├── test_inline_parser.py   # Inline parsing tests
│   ├── test_list_parser.py     # List parsing tests
│   ├── test_html_generator.py  # HTML generation tests
│   ├── test_integration.py     # End-to-end pipeline tests
│   └── fixtures/               # Test data files
│       ├── markdown_samples/   # Sample markdown inputs
│       └── expected_html/      # Expected HTML outputs
└── examples/                   # Usage examples and demos
    ├── basic_usage.py          # Simple API demonstration
    ├── custom_renderer.py      # Plugin interface example
    └── sample_documents/       # Example markdown files
```

This structure reflects several important architectural decisions:

Separation of Parsing Phases: Each major parsing component gets its own module (`block_parser.py`, `inline_parser.py`, `list_parser.py`). This makes it easy to work on one parsing phase without being distracted by others and enables focused unit testing.

Centralized Data Model: All AST node definitions live in `data_model.py`. This prevents circular imports and provides a single location for understanding the system's data structures. Any component that needs to create or manipulate AST nodes imports from this central module.

Utilities and Constants: Shared functionality like HTML escaping, indentation detection, and debug printing is centralized in `utils.py`. This prevents code duplication and ensures consistent behavior across components.

Comprehensive Testing: The test structure mirrors the module structure, with dedicated test files for each component plus integration tests that verify the complete pipeline. Fixture files separate test data from test logic, making tests easier to read and maintain.

> **Implementation Strategy**: Start by implementing `data_model.py` with basic AST node structures, then implement `preprocessor.py` and `utils.py` to establish the foundation. This enables incremental development where each subsequent milestone can build on solid, tested infrastructure.

The file organization also supports the **plugin architecture** mentioned in future extensions. Custom renderers can import the AST data model and implement alternative HTML generators without modifying the core parsing logic.

Implementation Guidance

Technology Recommendations

Component	Simple Option	Advanced Option	Rationale
Text Processing	Built-in `str` methods and `re` module	`regex` library with advanced features	Standard library sufficient for CommonMark compliance
Data Structures	Built-in `list` and `dict`	`collections.deque` for parser stacks	Standard containers handle AST and delimiter stacks adequately
File I/O	Built-in `open()` and text mode	`pathlib` for path handling	Simple file reading adequate, `pathlib` improves cross-platform compatibility
Testing Framework	Built-in `unittest`	`pytest` with fixtures	`pytest` provides cleaner test organization and better fixture support
CLI Interface	`argparse` for command-line parsing	`click` for advanced CLI features	`argparse` sufficient for basic file processing interface
Output Formatting	String concatenation and `join()`	Template engines like `jinja2`	String operations adequate for HTML generation, templates add complexity

For this beginner-level project, the **simple options** are strongly recommended. The standard library provides all necessary functionality, and additional dependencies would complicate the learning experience without providing significant benefits.

Core Infrastructure Starter Code

File: `utils.py` - Complete utility functions

PYTHON

```python
"""
Shared utilities and constants for markdown parsing.
Provides text processing utilities, HTML escaping, and debugging helpers.
"""

import re
from typing import List, Dict, Any

# HTML character escaping table
HTML_ESCAPE_TABLE = {
    '&': '&amp;',
    '<': '&lt;',
    '>': '&gt;',
    '"': '&quot;',
    "'": '&#x27;',
}

def normalize_line_endings(text: str) -> str:
    """Convert all line endings to Unix format (LF only)."""
    # Replace Windows CRLF first, then Mac CR
    return text.replace('\r\n', '\n').replace('\r', '\n')

def split_into_lines(text: str) -> List[str]:
    """Split text into lines, preserving empty lines."""
    return text.split('\n')

def is_blank_line(line: str) -> bool:
    """Check if line contains only whitespace characters."""
    return len(line.strip()) == 0

def get_indentation_level(line: str) -> int:
    """Count leading spaces in line. Tabs count as 4 spaces."""
    indent = 0
    for char in line:
```

```
        if char == ' ':
            indent += 1
        elif char == '\t':
            indent += 4
        else:
            break
    return indent

def escape_html(text: str) -> str:
    """Escape HTML special characters in text content."""
    result = text
    for char, entity in HTML_ESCAPE_TABLE.items():
        result = result.replace(char, entity)
    return result

def debug_print_ast(node: Any, indent: int = 0) -> None:
    """Print AST structure for debugging. Recursively shows node hierarchy."""
    prefix = "  " * indent
    node_info = f"{node.node_type}"
    if hasattr(node, 'block_type'):
        node_info += f" ({node.block_type})"
    elif hasattr(node, 'inline_type'):
        node_info += f" ({node.inline_type})"

    if hasattr(node, 'text_content') and node.text_content:
        # Truncate long text for readability
        text = node.text_content[:30] + "..." if len(node.text_content) > 30 else node.text_content
        node_info += f' "{text}"'

    print(f"{prefix}{node_info}")

    # Recursively print children
```

```
    if hasattr(node, 'children') and node.children:
        for child in node.children:
            debug_print_ast(child, indent + 1)
```

File: `data_model.py` - Complete AST node definitions

PYTHON

```python
"""
Abstract Syntax Tree node definitions for markdown document structure.
Defines the core data model used throughout the parsing pipeline.
"""

from typing import List, Optional, Dict, Any, Union
from enum import Enum


class NodeType(Enum):
    """Enumeration of all AST node types."""
    DOCUMENT = "document"
    BLOCK = "block"
    INLINE = "inline"
    TEXT = "text"


class BlockType(Enum):
    """Enumeration of block-level element types."""
    PARAGRAPH = "paragraph"
    HEADING = "heading"
    CODE_BLOCK = "code_block"
    BLOCKQUOTE = "blockquote"
    HORIZONTAL_RULE = "horizontal_rule"
    LIST = "list"
    LIST_ITEM = "list_item"


class InlineType(Enum):
    """Enumeration of inline element types."""
    STRONG = "strong"
    EMPHASIS = "emphasis"
    CODE = "code"
    LINK = "link"
    IMAGE = "image"
    LINE_BREAK = "line_break"
```

```
class ASTNode:
    """Base class for all AST nodes."""

    def __init__(self, node_type: NodeType, line_number: int = 0):
        self.node_type = node_type
        self.children: List['ASTNode'] = []
        self.parent: Optional['ASTNode'] = None
        self.line_number = line_number

    def add_child(self, child: 'ASTNode') -> None:
        """Add a child node and set its parent reference."""
        child.parent = self
        self.children.append(child)

    def remove_child(self, child: 'ASTNode') -> None:
        """Remove a child node and clear its parent reference."""
        if child in self.children:
            child.parent = None
            self.children.remove(child)

class BlockNode(ASTNode):
    """AST node for block-level elements."""

    def __init__(self, block_type: BlockType, line_number: int = 0):
        super().__init__(NodeType.BLOCK, line_number)
        self.block_type = block_type
        self.inline_content: str = ""
        self.block_attributes: Dict[str, Any] = {}

    def set_content(self, content: str) -> None:
        """Set the raw text content for this block."""
```

```
        self.inline_content = content

    def set_attribute(self, key: str, value: Any) -> None:
        """Set a block-specific attribute (e.g., heading level, language)."""
        self.block_attributes[key] = value

    def get_attribute(self, key: str, default: Any = None) -> Any:
        """Get a block-specific attribute with optional default."""
        return self.block_attributes.get(key, default)

class InlineNode(ASTNode):
    """AST node for inline formatting elements."""

    def __init__(self, inline_type: InlineType, line_number: int = 0):
        super().__init__(NodeType.INLINE, line_number)
        self.inline_type = inline_type
        self.text_content: str = ""
        self.formatting_attributes: Dict[str, Any] = {}

    def set_text(self, text: str) -> None:
        """Set the text content for this inline element."""
        self.text_content = text

    def set_attribute(self, key: str, value: Any) -> None:
        """Set an inline-specific attribute (e.g., URL, alt text)."""
        self.formatting_attributes[key] = value

    def get_attribute(self, key: str, default: Any = None) -> Any:
        """Get an inline-specific attribute with optional default."""
        return self.formatting_attributes.get(key, default)

class TextNode(ASTNode):
```

```
    """AST node for plain text content."""


    def __init__(self, text: str, line_number: int = 0):
        super().__init__(NodeType.TEXT, line_number)
        self.text_content = text


    def set_text(self, text: str) -> None:
        """Set the text content."""
        self.text_content = text
```

Core Parsing Logic Skeletons

File: `parser.py` - Main parser class skeleton

```python
"""
Main MarkdownParser class that coordinates the parsing pipeline.
This is the primary public interface for the markdown renderer.
"""

from typing import Optional
from .data_model import ASTNode, BlockNode, NodeType
from .preprocessor import Preprocessor
from .block_parser import BlockParser
from .inline_parser import InlineParser
from .html_generator import HTMLGenerator

class MarkdownParser:
    """Main parser that coordinates the two-phase parsing pipeline."""

    def __init__(self):
        self.block_parser = BlockParser()
        self.inline_parser = InlineParser()
        self.html_generator = HTMLGenerator()

    def parse_to_html(self, markdown_text: str) -> str:
        """
        Main parsing entry point. Converts markdown text to HTML.

        Args:
            markdown_text: Raw markdown input string

        Returns:
            Valid HTML string
        """
        # TODO 1: Use normalize_line_endings to standardize input
        # TODO 2: Use split_into_lines to break into line list
```

```
        # TODO 3: Call block_parser.parse(lines) to get block AST
        # TODO 4: Call inline_parser.parse(block_ast) to add inline elements
        # TODO 5: Call html_generator.generate(complete_ast) to produce HTML
        # TODO 6: Return the final HTML string
        pass


    def parse_file(self, file_path: str) -> str:
        """
        Parse markdown from a file path.

        Args:
            file_path: Path to markdown file

        Returns:
            HTML string
        """
        # TODO 1: Open and read file content with UTF-8 encoding
        # TODO 2: Call parse_to_html with file content
        # TODO 3: Handle FileNotFoundError and UnicodeDecodeError appropriately
        pass
```

File: `preprocessor.py` - Input processing skeleton

```python
"""
Input preprocessing for markdown text.
Handles normalization and basic text processing before parsing.
"""

from typing import List, Tuple
from .utils import normalize_line_endings, split_into_lines, is_blank_line, get_indentation_level

class LineInfo:
    """Metadata about a single line of input."""

    def __init__(self, content: str, line_number: int, is_blank: bool, indent_level: int):
        self.content = content
        self.line_number = line_number
        self.is_blank = is_blank
        self.indent_level = indent_level
        self.stripped_content = content.strip()

class Preprocessor:
    """Handles input normalization and line-level analysis."""

    def process_input(self, text: str) -> List[LineInfo]:
        """
        Process raw markdown input into analyzed lines.

        Args:
            text: Raw markdown text

        Returns:
            List of LineInfo objects with metadata
        """
        # TODO 1: Call normalize_line_endings on input text
```

```
        # TODO 2: Call split_into_lines to get line list
        # TODO 3: Create LineInfo object for each line with:
        #         - content (original line)
        #         - line_number (1-based)
        #         - is_blank (use is_blank_line utility)
        #         - indent_level (use get_indentation_level utility)
        # TODO 4: Return list of LineInfo objects
        pass
```

Milestone Checkpoints

After implementing the basic infrastructure (utils.py, data_model.py, parser.py):

Run the following verification:

```python
# Test basic infrastructure
from markdown_renderer.utils import escape_html, normalize_line_endings
from markdown_renderer.data_model import BlockNode, BlockType

# Verify HTML escaping
test_text = 'Hello & <world> "test"'
escaped = escape_html(test_text)
assert escaped == 'Hello &amp; &lt;world&gt; &quot;test&quot;'

# Verify line ending normalization
test_input = "Line 1\r\nLine 2\rLine 3\n"
normalized = normalize_line_endings(test_input)
assert normalized == "Line 1\nLine 2\nLine 3\n"

# Verify AST node creation
heading = BlockNode(BlockType.HEADING)
heading.set_attribute("level", 1)
heading.set_content("Test Heading")
assert heading.get_attribute("level") == 1
print("√ Basic infrastructure working correctly")
```

Expected behavior: All assertions should pass, demonstrating that the utility functions handle text processing correctly and AST nodes can store block-level content and attributes.

Signs something is wrong: If escaping doesn't handle all characters, check the `HTML_ESCAPE_TABLE` mapping. If line ending normalization fails, ensure Windows CRLF is processed before Mac CR. If AST node tests fail, verify the parent-child relationship management in `add_child`.

Language-Specific Implementation Hints

Python Text Processing Tips:

- Use `str.strip()` for whitespace removal, but remember it removes all whitespace characters including tabs and newlines
- `str.split('\n')` preserves empty strings for blank lines, which is essential for block boundary detection
- Regular expressions with `re.finditer()` provide both match content and position information useful for parsing
- List comprehensions with conditionals help filter lines: `[line for line in lines if not is_blank_line(line)]`

AST Manipulation Patterns:

- Always set parent references when adding children to enable upward traversal
- Use `isinstance(node, BlockNode)` for type checking instead of string comparison on `node_type`
- Consider using `collections.deque` for parser stacks if you need frequent insertion/removal at both ends
- Store line numbers in nodes during parsing - they're invaluable for error reporting and debugging

Error Handling Strategy:

- Catch `UnicodeDecodeError` when reading files and provide helpful error messages
- Use `try/except` around file operations but let parsing errors propagate for debugging
- Consider using `logging` module for debug output rather than print statements
- Validate AST structure with assertions during development, but remove them for production

> **Development Workflow Tip**: Implement and test each utility function independently before building the parser components. Use `python -m pytest tests/test_utils.py -v` to verify individual functions work correctly before integrating them into the larger parsing pipeline.

Data Model

> **Milestone(s):** Milestone 1: Block Elements, Milestone 2: Inline Elements, Milestone 3: Lists, Milestone 4: HTML Generation

The Document Tree Mental Model

Think of a markdown document as a family tree where each person (element) knows their role in the family hierarchy. Just as a family tree has generations—grandparents, parents, children—a markdown document has structural levels. The document root is like the family patriarch, containing major sections (headings, paragraphs, lists) as its children, which in turn contain their own children (bold text, links, images within paragraphs).

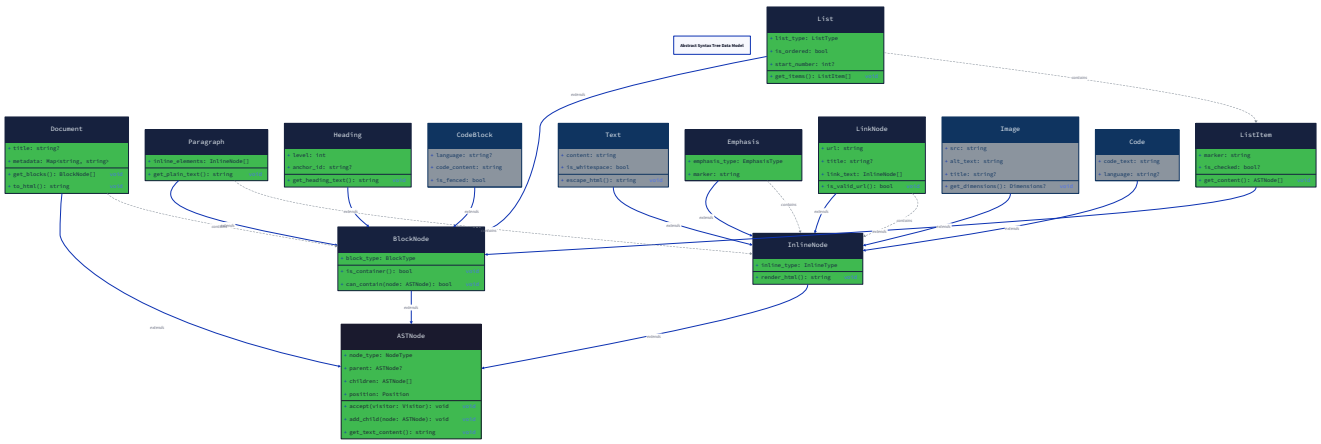
Unlike a flat text file where everything is just characters in sequence, our parsed representation maintains these relationships explicitly. Each node in the tree knows its parent (the element that contains it), its children (the elements it contains), and its

siblings (elements at the same level). This hierarchical structure makes it straightforward to apply consistent formatting rules, validate nesting constraints, and generate properly structured HTML output.

The key insight is that markdown parsing transforms linear text into a structured tree where semantic relationships are explicit rather than implicit. A paragraph doesn't just contain the text "This is **bold** text"—it contains a sequence of child elements: a text node "This is ", a strong emphasis node containing "bold", and another text node " text". This explicitness enables precise control over HTML generation and makes complex transformations tractable.

Abstract Syntax Tree Nodes

The Abstract Syntax Tree (AST) serves as the central data structure throughout our parsing pipeline. Every element in the markdown document, from the document root down to individual words, is represented as a node in this tree. The AST captures not just the content of each element but its semantic meaning, formatting attributes, and relationship to other elements.



Base AST Node Structure

The foundation of our AST is the `ASTNode` base type, which provides the common interface and shared properties that all document elements require. Every node in the tree, regardless of its specific type or content, inherits these fundamental characteristics.

Field	Type	Description
`node_type`	`NodeType`	Enum value identifying the general category of this node (DOCUMENT, BLOCK, INLINE, TEXT)
`children`	`List[ASTNode]`	Ordered list of child nodes contained within this element
`parent`	`ASTNode`	Reference to the parent node containing this element (None for document root)
`line_number`	`int`	Source line number where this element begins in the original markdown text

The `node_type` field enables type-safe traversal and processing—when the HTML generator encounters a node, it can immediately dispatch to the appropriate rendering logic without expensive type checking. The `children` list maintains document order, which is crucial for elements like paragraphs where the sequence of text and formatting matters. The `parent` reference enables bidirectional tree navigation, supporting operations like "find the containing list to determine nesting level" or "check if this emphasis is inside a link".

Line number tracking serves multiple purposes beyond debugging. During error reporting, we can provide precise source locations for malformed elements. During incremental parsing (a future extension), we can map document edits back to specific AST subtrees for efficient re-parsing.

Block-Level Node Structure

Block-level elements represent the document's structural backbone—paragraphs, headings, code blocks, lists, and other elements that establish the document's major sections. These elements typically span multiple lines and cannot appear inside other block elements (with specific exceptions like list items containing paragraphs).

Field	Type	Description
`block_type`	`BlockType`	Specific type of block element (PARAGRAPH, HEADING, CODE_BLOCK, etc.)
`inline_content`	`str`	Raw text content before inline parsing (e.g., "This is **bold** text")
`block_attributes`	`Dict[str, Any]`	Element-specific metadata (heading level, code language, list type)

The `BlockNode` extends `ASTNode` with block-specific functionality. The `inline_content` field stores the raw text before inline parsing occurs—this supports our two-phase parsing approach where we first identify block structure, then parse inline formatting within each block. For a heading, this might be "# Chapter **One**: Introduction" before we parse the emphasis within the heading text.

The `block_attributes` dictionary provides flexibility for element-specific properties without requiring separate node types for every variation. A heading node stores its level (1-6) as `{"level": 2}`, while a code block stores its language hint as `{"language": "python", "is_fenced": true}`. This approach maintains type safety while avoiding an explosion of specialized node classes.

> **Key Design Insight**: Separating block structure from inline content enables parallel processing in future extensions—we could parse multiple blocks' inline content concurrently once the document structure is established.

Inline Element Node Structure

Inline elements provide formatting and semantic markup within block-level containers. Unlike block elements, inline elements can nest arbitrarily—emphasis can contain links, links can contain code spans (with restrictions), and complex combinations create rich text formatting.

Field	Type	Description
`inline_type`	`InlineType`	Specific type of inline element (STRONG, EMPHASIS, CODE, LINK, IMAGE, LINE_BREAK)
`text_content`	`str`	The actual text content for this inline element
`formatting_attributes`	`Dict[str, Any]`	Element-specific properties (URL for links, alt text for images)

The `InlineNode` specializes `ASTNode` for inline formatting elements. The `text_content` field holds the processed text—for emphasis, this is the text between the markers ("bold" from "**bold**"). For links, this is the link text, while the URL is stored in `formatting_attributes` as `{"url": "https://example.com", "title": "Optional title"}`.

This separation between content and attributes proves crucial during HTML generation. The generator can apply consistent escaping to all text content while handling attributes according to their specific requirements (URLs need different escaping than alt text).

Text Node Structure

Text nodes represent the leaves of our AST—the actual words, punctuation, and whitespace that form the document's readable content. These nodes contain no child elements and require minimal metadata.

Text nodes inherit the base `ASTNode` structure but typically have empty `children` lists and use the `text_content` for their actual content. While we could define a specialized `TextNode` type, the base `ASTNode` with `node_type = TEXT` provides sufficient functionality while maintaining interface consistency.

Node Type Enumerations

The type system uses enumerations to provide compile-time safety and clear semantic boundaries between different element categories.

NodeType Enumeration:

Value	Purpose	Examples
`DOCUMENT`	Root container for entire document	Document root node
`BLOCK`	Block-level structural elements	Paragraphs, headings, code blocks
`INLINE`	Inline formatting elements	Emphasis, links, inline code
`TEXT`	Leaf nodes containing actual text	Word sequences, punctuation

BlockType Enumeration:

Value	Purpose	Attributes Used
`PARAGRAPH`	Regular paragraph text	None
`HEADING`	Section headings	`level` (1-6)
`CODE_BLOCK`	Preformatted code	`language`, `is_fenced`
`BLOCKQUOTE`	Quoted text sections	`nesting_level`
`HORIZONTAL_RULE`	Section dividers	None
`LIST`	Ordered/unordered lists	`list_type` (ordered/unordered), `start_number`
`LIST_ITEM`	Individual list items	`marker_type`, `is_tight`

InlineType Enumeration:

Value	Purpose	Attributes Used
`STRONG`	Bold/strong emphasis	None
`EMPHASIS`	Italic/emphasis	None
`CODE`	Inline code spans	None
`LINK`	Hyperlinks	`url`, `title`
`IMAGE`	Embedded images	`url`, `alt_text`, `title`
`LINE_BREAK`	Hard line breaks	`break_type` (soft/hard)

Parser Context and State

The parsing process requires sophisticated state management to handle markdown's context-sensitive rules and maintain consistency across the two-phase parsing pipeline. Our state structures capture both the current parsing context and the accumulated results as parsing progresses.

Primary Parser State

The main parser coordinates the overall parsing process and maintains references to specialized parsers for different document elements. This centralized state enables consistent behavior and shared utility functions across parsing phases.

Field	Type	Description
`block_parser`	`BlockParser`	Specialized parser for block-level elements
`inline_parser`	`InlineParser`	Specialized parser for inline formatting elements
`html_generator`	`HTMLGenerator`	Component responsible for AST-to-HTML conversion

The `MarkdownParser` serves as the facade for the entire parsing system, orchestrating the flow from raw markdown text through block parsing, inline parsing, and final HTML generation. This design enables clean separation of concerns while providing a simple external interface.

Line Processing Context

Before any structural parsing begins, the raw markdown text undergoes preprocessing to normalize line endings, analyze indentation, and identify blank lines. The `LineInfo` structure captures this analysis for each source line.

Field	Type	Description
`content`	`str`	The line's text content with normalized whitespace handling
`line_number`	`int`	One-based line number in the original source document
`is_blank`	`bool`	True if the line contains only whitespace characters
`indent_level`	`int`	Number of leading spaces (tabs converted to spaces)

The preprocessing phase transforms raw input into a list of `LineInfo` objects that subsequent parsing phases consume. This preprocessing handles cross-platform compatibility (normalizing Windows/Mac line endings), establishes consistent indentation measurement, and pre-identifies blank lines that serve as block separators.

The `indent_level` calculation converts tabs to spaces using a configurable tab width (typically 4 spaces) and handles mixed indentation consistently. This preprocessing eliminates platform-specific edge cases from the core parsing logic.

Block Parser State

During block-level parsing, the parser maintains state about the current parsing context, including nesting levels for elements like blockquotes and lists, and accumulated content for multi-line elements like paragraphs.

Block parser state includes several key components that track the parsing progress and accumulated results. The parser maintains a stack of open block elements to handle nested structures like blockquotes containing lists. It tracks the current line position within the input and maintains buffers for accumulating multi-line content.

The block parser uses a state machine approach where each line can trigger transitions between parsing modes. When processing a paragraph, each subsequent non-blank line accumulates into the paragraph content until a blank line or new block element triggers paragraph closure and inline parsing of the accumulated content.

For list processing, the block parser maintains additional state tracking current list nesting levels, list types at each level, and tight versus loose list formatting. This state enables proper handling of complex nested list structures while maintaining CommonMark compliance.

Inline Parser State

Inline parsing requires sophisticated state management to handle nested elements and delimiter matching. The inline parser maintains a delimiter stack to track opening markers (asterisks, underscores, backticks) and match them with corresponding closing markers according to CommonMark precedence rules.

The delimiter stack tracks not just the marker characters but their positions, surrounding context, and nesting relationships. When processing text like "**bold *italic* text**", the parser must correctly identify that the asterisks for italic are nested within the asterisks for bold, requiring careful stack management.

The inline parser also maintains context about whether it's currently inside certain elements that restrict further nesting. For example, code spans cannot contain other inline formatting, so the parser suppresses emphasis processing while inside backtick-delimited code.

Link parsing requires special state management because link text can contain other inline formatting, but the URL portion cannot. The parser tracks whether it's currently inside link text versus link URLs and adjusts its behavior accordingly.

> **Architecture Decision: Stateful vs Stateless Parsing**
>
> - **Context**: Choose between maintaining parser state versus pure functional parsing
> - **Options Considered**:
>   - Stateful parsers with mutable context objects
>   - Pure functional parsers passing immutable state
>   - Hybrid approach with immutable state but mutable result accumulation
> - **Decision**: Stateful parsers with carefully controlled mutable state
> - **Rationale**: Markdown's context-sensitive rules (especially for lists and emphasis) create complex state dependencies that are clearer to express with mutable state. Functional approaches would require threading complex state through every parsing function.
> - **Consequences**: Enables more readable parsing logic and better performance, but requires careful state management to avoid bugs from unexpected mutations.

AST Construction and Manipulation

The AST provides a comprehensive interface for constructing and manipulating the document tree during parsing. These operations maintain tree invariants and provide safe access to the hierarchical structure.

Node Construction Interface

Method	Parameters	Returns	Description
`add_child`	`child: ASTNode`	`None`	Appends child node and sets its parent reference
`set_content`	`content: str`	`None`	Sets the text content for leaf nodes
`set_attribute`	`key: str, value: Any`	`None`	Stores element-specific attribute
`get_attribute`	`key: str, default: Any`	`Any`	Retrieves attribute value or default

The `add_child` method maintains bidirectional parent-child relationships automatically. When a child is added to a parent, the child's parent reference is updated, and if the child was previously attached to a different parent, it's removed from the old parent's children list. This ensures tree consistency without manual reference management.

The attribute methods provide type-safe access to element-specific properties. Rather than exposing the attribute dictionary directly, these methods enable validation and default value handling. For example, `get_attribute("level", 1)` retrieves a heading's level with a sensible default.

Tree Traversal Interface

Method	Parameters	Returns	Description
`depth_first_walk`	`visitor: function`	`None`	Visits all nodes in depth-first order
`find_nodes_by_type`	`node_type: NodeType`	`List[ASTNode]`	Returns all descendant nodes of specified type
`get_text_content`	None	`str`	Returns concatenated text from all descendant text nodes

Tree traversal methods support common operations needed during HTML generation and analysis. The `depth_first_walk` method accepts a visitor function that's called for each node, enabling custom processing without exposing tree structure details.

The `get_text_content` method provides a convenient way to extract all text from a subtree, useful for generating alt text, computing heading anchor text, or extracting link text for processing.

Common Pitfalls

⚠️ **Pitfall: Circular Parent References** When manually constructing AST nodes, developers sometimes create circular references by setting parent pointers incorrectly. This leads to infinite loops during tree traversal. Always use the `add_child` method rather than setting parent/children references manually—it maintains consistency automatically.

⚠️ **Pitfall: Modifying Children During Traversal** Modifying a node's children list while iterating over it (for example, during tree transformation) can skip nodes or cause index errors. Create a copy of the children list before iteration, or use reverse iteration when removing nodes.

⚠️ **Pitfall: Inconsistent Node Types** Setting a node's `node_type` to `BLOCK` but using `InlineType` values in type-specific fields creates confusion during processing. The type hierarchy must be consistent—`BlockNode` instances should only use `BlockType` values, and the base `node_type` should match the specialized type field.

⚠️ **Pitfall: Missing Line Number Tracking** Failing to set line numbers during parsing makes debugging extremely difficult. Users report "the heading is malformed" but without line numbers, finding the problem in large documents becomes painful. Always propagate line numbers from `LineInfo` to AST nodes.

⚠️ **Pitfall: Deep Copying Node References** When cloning or serializing AST nodes, naive deep copying creates duplicate parent/child references that break tree structure. Implement custom cloning that reconstructs relationships rather than copying references directly.

Implementation Guidance

The AST implementation requires careful attention to memory management, type safety, and tree consistency. The following guidance provides practical approaches for implementing the data model in Python.

Technology Recommendations

Component	Simple Option	Advanced Option
Node Classes	Python dataclasses with inheritance	Custom classes with `__slots__` for memory efficiency
Type Safety	Python enums for node types	Python typing with Union types and type guards
Tree Operations	Recursive methods with explicit stack management	Generator-based traversal to avoid stack overflow
Attribute Storage	Simple dictionaries	Typed attribute classes for validation

Recommended File Structure

```
markdown_renderer/
  ast/
    __init__.py              ← Export public AST interface
    nodes.py                 ← Core node class definitions
    traversal.py             ← Tree traversal utilities
    builder.py               ← High-level AST construction helpers
  parser/
    context.py               ← Parser state and context classes
    line_info.py             ← Line preprocessing utilities
```

This structure separates AST data structures from parsing logic while keeping related functionality grouped. The `ast` package can be imported independently for applications that work with pre-built trees.

Infrastructure Starter Code

Complete LineInfo Processing (`parser/line_info.py`):

```python
from dataclasses import dataclass

from typing import List

import re


@dataclass

class LineInfo:

    content: str

    line_number: int

    is_blank: bool

    indent_level: int


class Preprocessor:

    TAB_WIDTH = 4



    def normalize_line_endings(self, text: str) -> str:

        """Convert all line endings to Unix format (\n)."""

        return text.replace('\r\n', '\n').replace('\r', '\n')



    def split_into_lines(self, text: str) -> List[str]:

        """Split text into lines, preserving empty lines."""

        return text.split('\n')



    def is_blank_line(self, line: str) -> bool:

        """Check if line contains only whitespace."""

        return len(line.strip()) == 0



    def get_indentation_level(self, line: str) -> int:

        """Count leading spaces, converting tabs to spaces."""

        indent = 0

        for char in line:

            if char == ' ':

                indent += 1
```

```
            elif char == '\t':
                indent += self.TAB_WIDTH
            else:
                break
        return indent


    def process_input(self, text: str) -> List[LineInfo]:
        """Convert raw markdown text into analyzed line information."""
        normalized = self.normalize_line_endings(text)
        lines = self.split_into_lines(normalized)

        line_infos = []
        for i, line in enumerate(lines):
            line_info = LineInfo(
                content=line,
                line_number=i + 1,
                is_blank=self.is_blank_line(line),
                indent_level=self.get_indentation_level(line)
            )
            line_infos.append(line_info)

        return line_infos
```

HTML Escaping Utilities (`ast/nodes.py`):

```python
from typing import Dict


# HTML character escaping table for safe output generation

HTML_ESCAPE_TABLE = {

    '&': '&amp;',

    '<': '&lt;',

    '>': '&gt;',

    '"': '&quot;',

    "'": '&#x27;'

}


def escape_html(text: str) -> str:

    """Escape HTML special characters to prevent XSS and ensure valid output."""

    for char, entity in HTML_ESCAPE_TABLE.items():

        text = text.replace(char, entity)

    return text
```

Core Logic Skeleton Code

Base AST Node Implementation (`ast/nodes.py`):

```python
from enum import Enum

from typing import List, Optional, Any, Dict

from dataclasses import dataclass, field


class NodeType(Enum):

    DOCUMENT = "document"

    BLOCK = "block"

    INLINE = "inline"

    TEXT = "text"


class BlockType(Enum):

    PARAGRAPH = "paragraph"

    HEADING = "heading"

    CODE_BLOCK = "code_block"

    BLOCKQUOTE = "blockquote"

    HORIZONTAL_RULE = "horizontal_rule"

    LIST = "list"

    LIST_ITEM = "list_item"


class InlineType(Enum):

    STRONG = "strong"

    EMPHASIS = "emphasis"

    CODE = "code"

    LINK = "link"

    IMAGE = "image"

    LINE_BREAK = "line_break"


@dataclass

class ASTNode:

    node_type: NodeType

    children: List['ASTNode'] = field(default_factory=list)

    parent: Optional['ASTNode'] = None

    line_number: int = 0
```

```python
def add_child(self, child: 'ASTNode') -> None:
    """Add child node and maintain bidirectional parent-child relationship."""
    # TODO 1: Remove child from its current parent if it has one
    # TODO 2: Add child to this node's children list
    # TODO 3: Set child's parent reference to this node
    # Hint: Check if child.parent exists before removing from old parent
    pass

def set_content(self, content: str) -> None:
    """Set text content for leaf nodes."""
    # TODO 1: Store content in appropriate field based on node type
    # TODO 2: For text nodes, content goes directly in text field
    # TODO 3: For block nodes, content goes in inline_content for later inline parsing
    # Hint: This method should behave differently for different node types
    pass

def set_attribute(self, key: str, value: Any) -> None:
    """Set element-specific attribute."""
    # TODO 1: Ensure attributes dictionary exists
    # TODO 2: Store key-value pair in appropriate attributes field
    # TODO 3: Handle validation for known attribute keys if desired
    pass

def get_attribute(self, key: str, default: Any = None) -> Any:
    """Get element-specific attribute with optional default."""
    # TODO 1: Check if attributes dictionary exists
    # TODO 2: Return attribute value if key exists
    # TODO 3: Return default value if key doesn't exist
    pass
```

```python
@dataclass
class BlockNode(ASTNode):
    block_type: BlockType = BlockType.PARAGRAPH
    inline_content: str = ""
    block_attributes: Dict[str, Any] = field(default_factory=dict)

    def __post_init__(self):
        # Ensure node_type is consistent with being a block node
        self.node_type = NodeType.BLOCK

@dataclass
class InlineNode(ASTNode):
    inline_type: InlineType = InlineType.EMPHASIS
    text_content: str = ""
    formatting_attributes: Dict[str, Any] = field(default_factory=dict)

    def __post_init__(self):
        # Ensure node_type is consistent with being an inline node
        self.node_type = NodeType.INLINE
```

AST Traversal Utilities (`ast/traversal.py`):

PYTHON

```python
from typing import Callable, List

from .nodes import ASTNode, NodeType


def debug_print_ast(node: ASTNode, indent: int = 0) -> None:

    """Print AST structure for debugging purposes."""

    # TODO 1: Print current node with indentation showing tree depth

    # TODO 2: Include node type, line number, and key attributes

    # TODO 3: Recursively print all child nodes with increased indentation

    # TODO 4: For text nodes, show truncated content

    # TODO 5: For block/inline nodes, show their specific type

    # Hint: Use " " * indent for indentation, truncate long content

    pass


def depth_first_walk(node: ASTNode, visitor: Callable[[ASTNode], None]) -> None:

    """Visit all nodes in depth-first order."""

    # TODO 1: Call visitor function on current node

    # TODO 2: Recursively walk all child nodes

    # TODO 3: Handle any exceptions from visitor gracefully

    # Hint: Visit parent before children for depth-first pre-order

    pass


def find_nodes_by_type(root: ASTNode, target_type: NodeType) -> List[ASTNode]:

    """Find all descendant nodes of specified type."""

    results = []



    def collector(node: ASTNode) -> None:

        # TODO 1: Check if current node matches target_type

        # TODO 2: Add matching nodes to results list

        pass



    # TODO 3: Use depth_first_walk with collector function

    # TODO 4: Return accumulated results
```

```
    return results


def get_text_content(node: ASTNode) -> str:
    """Extract all text content from node and descendants."""
    text_parts = []


    def text_collector(node: ASTNode) -> None:
        # TODO 1: If node is text type, add its content to text_parts
        # TODO 2: For other node types, extract text from appropriate fields
        # TODO 3: Handle inline nodes with text_content field
        # TODO 4: Handle block nodes with inline_content field
        pass


    # TODO 5: Walk tree collecting text
    # TODO 6: Join all text parts and return
    return ""
```

Language-Specific Implementation Notes

Memory Efficiency in Python:

- Use `__slots__` in node classes if memory usage becomes an issue with large documents
- Consider `weakref` for parent references to avoid circular reference memory leaks
- Use generators for tree traversal to avoid building large intermediate lists

Type Safety:

- Use `typing.Union` to create precise type hints for methods that accept multiple node types
- Consider using `typing.Protocol` to define interfaces rather than concrete inheritance
- Use `isinstance()` checks with enum values for type-safe node processing

Performance Considerations:

- Cache frequently-accessed attributes like text content extraction
- Use iterative traversal instead of recursion for very deep document trees
- Consider lazy evaluation for expensive operations like full-text search

Milestone Checkpoint

After implementing the AST data model, verify functionality with these tests:

1. **Node Construction Test**: Create nodes of each type and verify all fields are properly initialized
2. **Parent-Child Relationship Test**: Add children to parents and verify bidirectional references

3. **Tree Traversal Test**: Build a small tree and verify depth-first traversal visits nodes in correct order
4. **Attribute Management Test**: Set and retrieve attributes on different node types
5. **Line Processing Test**: Process sample markdown text and verify `LineInfo` objects have correct properties

Expected Behavior:

- `debug_print_ast()` should show proper tree structure with indentation
- Parent-child relationships should be consistent after `add_child()` calls
- Line preprocessing should correctly identify blank lines and indentation levels
- HTML escaping should convert `<script>` to `&lt;script&gt;`

Common Issues to Check:

- Circular references causing infinite loops in traversal
- Missing line numbers making debugging difficult
- Inconsistent node types between base and specialized fields
- Memory leaks from strong reference cycles in parent-child relationships

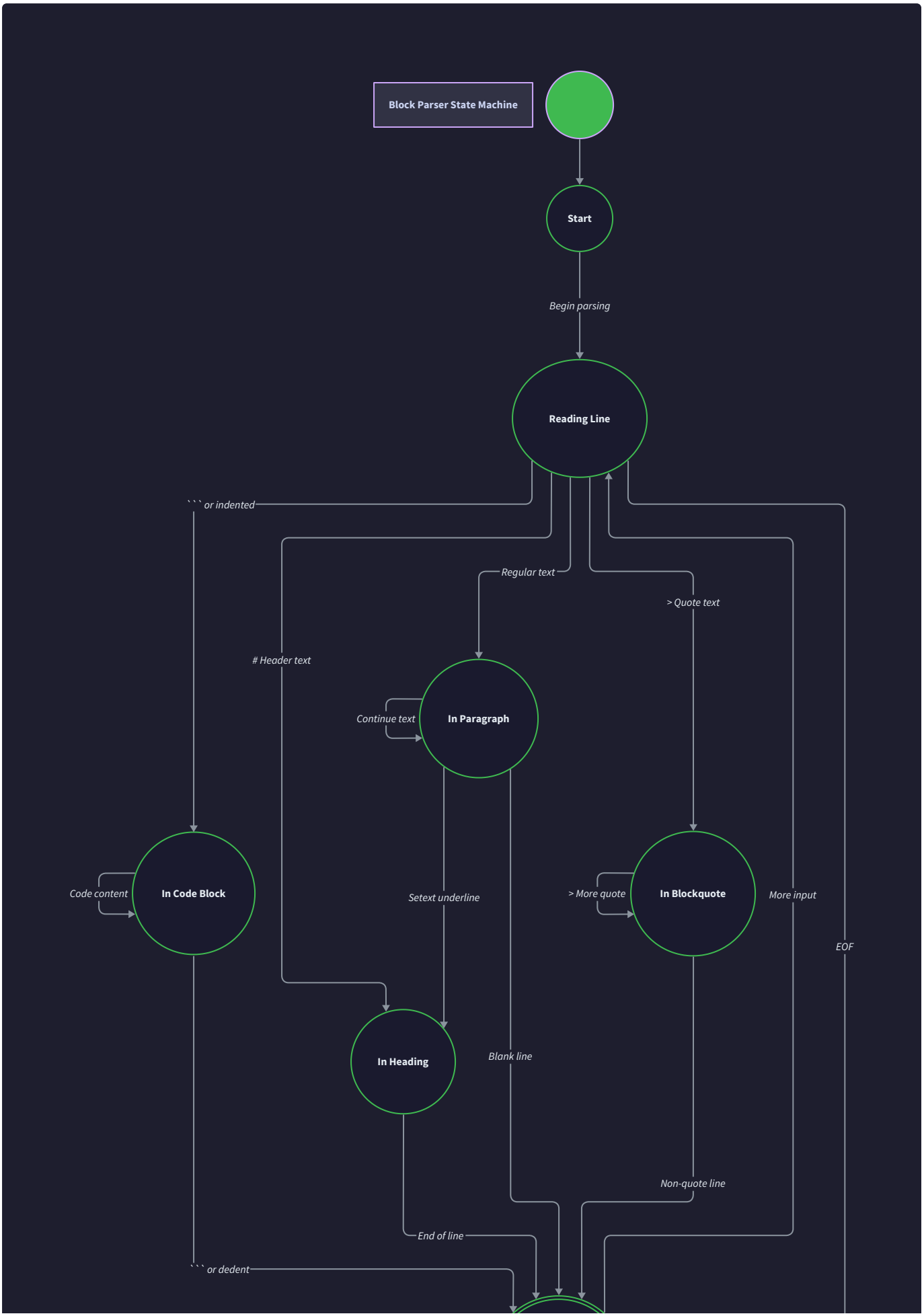
Block Parser Design

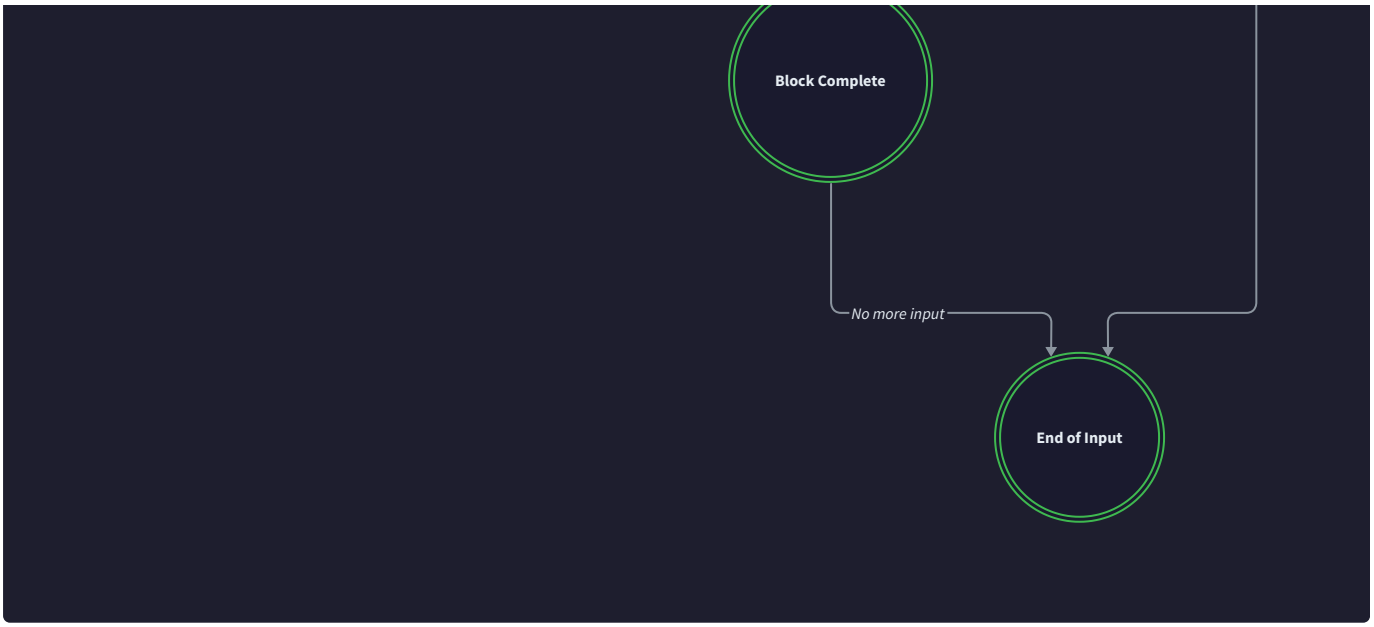
> **Milestone(s):** Milestone 1: Block Elements

The Building Blocks Mental Model

Think of block parsing like organizing a messy stack of papers into clearly labeled folders. When you process a pile of documents, you first separate them into major categories—reports go in one folder, memos in another, code printouts in a third. You don't worry about the formatting within each document (bold text, underlined words) until after you've sorted them into the right folders. Block parsing works the same way: it takes the raw lines of markdown text and groups them into structural containers like headings, paragraphs, code blocks, and blockquotes. Each block becomes a clearly defined section that can later be filled with inline formatting details.

The key insight is that block-level structure provides the skeleton of the document, while inline elements provide the flesh. You must build the skeleton first before you can properly attach the details. This two-phase approach prevents the parser from getting confused when it encounters complex nested structures like a bold link inside a blockquote within a list item.


Block Parser State Machine
Start
Begin parsing
Reading Line
``` or indented
Regular text
> Quote text
# Header text
Continue text
In Paragraph
Code content
In Code Block
Setext underline
> More quote
In Blockquote
More input
EOF
In Heading
Blank line
Non-quote line
End of line
``` or dedent




## Block Parsing Algorithm

The block parsing algorithm operates as a **state machine** that processes the markdown text line by line, maintaining context about what type of block is currently being built. The parser must handle the fact that some blocks span multiple lines (paragraphs, code blocks) while others are determined by a single line (headings, horizontal rules). Additionally, some blocks require lookahead to determine their type (distinguishing between a paragraph and a heading with Setext underlines).

The algorithm follows this sequential process:

1. **Input Preprocessing**: The raw markdown text is normalized by converting all line endings to Unix format ( `\n` ) and splitting into individual lines while preserving line numbers for error reporting. Each line is wrapped in a `LineInfo` object that captures the original content, line number, whether it's blank, and its indentation level.
2. **Block Detection Loop**: The parser iterates through the processed lines, maintaining a current parsing state. For each line, it first checks if the current block type can continue (for multi-line blocks like paragraphs or code blocks). If the current block cannot continue, it finalizes the current block and attempts to start a new block based on the line's characteristics.
3. **Block Type Recognition**: The parser uses a priority-ordered sequence of pattern matching to determine block types. Headings are checked first (both ATX-style with `#` prefixes and potential Setext-style with underline lookahead), followed by code blocks (both fenced with triple backticks and indented), then blockquotes (lines starting with `>` ), horizontal rules (three or more dashes or asterisks), and finally defaulting to paragraph blocks.
4. **Multi-line Block Handling**: For blocks that span multiple lines, the parser maintains state about the current block being constructed. Paragraph blocks accumulate consecutive non-blank lines until encountering a blank line or a line that starts a new block type. Code blocks have different continuation rules depending on whether they're fenced (continue until closing fence) or indented (continue while indentation is maintained).
5. **Blank Line Processing**: Blank lines serve as block separators in most contexts, but their handling depends on the current block type. They terminate paragraph blocks and separate loose list items, but are preserved within fenced code blocks and ignored within indented code blocks after the required indentation is stripped.
6. **Block Finalization**: When a block is completed (either by encountering a new block type or reaching the end of input), the parser creates a `BlockNode` with the appropriate `BlockType` , stores the raw content that will later be processed for inline elements, and adds it to the growing AST.
7. **Lookahead Handling**: Some block types require examining subsequent lines to make parsing decisions. Setext headings are the primary example—a line of text followed by a line of equals signs or dashes indicates a heading, not two separate

paragraphs. The parser implements limited lookahead by peeking at the next line when the current line could potentially be part of a Setext heading.

8. **Context Preservation**: Throughout parsing, the algorithm maintains context about nesting levels (important for lists and blockquotes), indentation tracking, and the parent-child relationships needed to build the hierarchical AST structure.

The parser uses this state machine approach rather than attempting to parse all block types simultaneously with complex regular expressions. This provides better error recovery, clearer debugging, and easier extension to support additional block types.

### Block Parser Architecture Decisions

The block parser must make several critical architectural choices that significantly impact both the implementation complexity and the system's extensibility. Each decision involves trade-offs between parsing accuracy, performance, and maintainability.

**Decision: State Machine vs. Regular Expression Matching**

- **Context**: Block parsing can be implemented either as a state machine that tracks parsing context across lines, or as a collection of regular expressions that pattern-match individual lines
- **Options Considered**: Stateful line-by-line processing, regex-based pattern matching, hybrid approach with regex patterns within state machine
- **Decision**: State machine with regex patterns for individual line recognition
- **Rationale**: State machines provide better context tracking for multi-line blocks and cleaner error recovery, while regex patterns excel at recognizing specific line formats like headings and horizontal rules
- **Consequences**: Enables robust handling of complex cases like nested blockquotes and provides clear extension points for new block types, but requires more complex state management

| Approach | Pros | Cons | Chosen? |
|---|---|---|---|
| Pure State Machine | Clean context tracking, excellent error recovery, easy debugging | Complex state transitions, harder to modify individual patterns | No |
| Pure Regex | Simple pattern matching, easy to modify individual rules | No context between lines, poor error handling, brittle | No |
| Hybrid State Machine + Regex | Context tracking + flexible patterns, maintainable | Moderate complexity in state management | Yes |

**Decision: Lookahead Strategy for Setext Headings**

- **Context**: Setext headings require examining the next line to determine if a text line should become a heading, but lookahead complicates the single-pass parsing algorithm
- **Options Considered**: Single-line lookahead buffer, two-pass preprocessing, ignore Setext headings entirely
- **Decision**: Limited single-line lookahead with buffering
- **Rationale**: Setext headings are common enough to support, but full two-pass parsing adds unnecessary complexity for this single use case
- **Consequences**: Requires maintaining a one-line buffer and special handling in the main parsing loop, but keeps the parser mostly single-pass

| Lookahead Strategy | Pros | Cons | Chosen? |
|---|---|---|---|
| No Lookahead | Simplest parsing, single-pass | Cannot handle Setext headings | No |
| Single-line Buffer | Handles Setext headings, mostly single-pass | Slightly more complex state | Yes |
| Full Two-pass | Handles all complex cases | Much more complex, performance overhead | No |

**Decision: Block Content Storage Format**

- **Context**: Blocks need to store their raw content for later inline processing, but the storage format affects both memory usage and inline parsing complexity
- **Options Considered**: Store as single concatenated string, maintain array of original lines, store preprocessed text with whitespace normalized
- **Decision**: Store as array of original line strings with metadata
- **Rationale**: Preserves original formatting for accurate inline parsing while maintaining line-level information needed for error reporting and debugging
- **Consequences**: Higher memory usage but better debugging information and more accurate inline parsing, especially for code blocks where whitespace is significant

| Storage Format | Pros | Cons | Chosen? |
|---|---|---|---|
| Single String | Low memory, simple | Loses line boundaries, poor debugging | No |
| Line Array | Preserves formatting, good debugging | Higher memory usage | Yes |
| Preprocessed | Normalized format | May lose important whitespace | No |

**Decision: Error Recovery Strategy**

- **Context**: When encountering malformed block syntax, the parser must decide whether to fail, skip the problematic content, or attempt graceful recovery
- **Options Considered**: Fail on any syntax error, skip malformed blocks entirely, treat unrecognized syntax as paragraph text
- **Decision**: Graceful degradation by treating unrecognized blocks as paragraphs
- **Rationale**: Markdown philosophy emphasizes readability even when formatting is imperfect, so the parser should produce reasonable output from imperfect input
- **Consequences**: Users get output even from malformed input, but may not notice formatting errors that should be corrected

The block parser implements these decisions through a `BlockParser` class that maintains parsing state and delegates to specialized recognition functions for each block type. The state machine tracks the current block being built, while individual regex patterns handle line-level recognition within the appropriate state context.

## Block Parsing Common Pitfalls

Block parsing presents several subtle challenges that frequently trip up developers implementing their first markdown parser. These pitfalls often stem from the interaction between line-level pattern matching and multi-line context tracking.

**⚠️ Pitfall: Forgetting Setext Heading Lookahead**

Many developers implement ATX headings (lines starting with `#`) correctly but forget that Setext headings require examining the following line. They parse each line independently and end up treating "Title\n====" as a paragraph containing "Title" followed by another paragraph containing "====", instead of recognizing it as a level-1 heading.

The issue occurs because single-line parsing cannot distinguish between a standalone line of text and the first line of a Setext heading. The parser must peek at the next line to check for underline characters (`=` for h1, `-` for h2) before deciding how to classify the current line.

To avoid this, implement a one-line lookahead buffer in your main parsing loop. When you encounter a non-blank line that could be a heading, check if the next line consists entirely of `=` or `-` characters. Only commit to creating a paragraph block after confirming the next line is not a Setext underline.

**⚠️ Pitfall: Incorrect Indented Code Block Detection**

Indented code blocks require exactly four spaces of indentation, but many parsers incorrectly handle mixed tabs and spaces or fail to distinguish between indented code and normal paragraph continuation. A common mistake is treating any indented line as code, which breaks when users indent regular paragraphs for visual formatting.

The problem is compounded by the fact that indented code blocks must be preceded by a blank line (or start of document) to distinguish them from indented continuation of previous blocks like list items or blockquotes. Without this check, indented text within lists gets incorrectly parsed as code blocks.

Implement precise indentation checking by normalizing tabs to spaces early in preprocessing, then checking for exactly four spaces (or one tab) at the line start. Additionally, verify that indented code blocks are preceded by blank lines or other block boundaries, not continuation of existing blocks.

**⚠️ Pitfall: Fenced Code Block Language Hints**

Fenced code blocks can specify a language identifier immediately after the opening triple backticks (e.g., "```python"), but many parsers either ignore this information entirely or fail to handle edge cases like languages with special characters or extra whitespace.

The language hint affects HTML generation because it should be included in the CSS class of the generated `<code>` element, but parsers often store only the content between the fences without preserving the language metadata. This breaks syntax highlighting and other language-specific processing.

Extract the language hint during fenced code block parsing by capturing everything after the opening fence until the first whitespace or newline. Store this as a block attribute that the HTML generator can access later. Handle the case where no language is specified by storing an empty string rather than null.

**⚠️ Pitfall: Blockquote Nesting and Lazy Continuation**

Blockquote parsing has two subtle complications: nested blockquotes (lines with multiple `>` prefixes) and lazy continuation (subsequent lines without `>` that continue the blockquote content). Many parsers handle only simple single-level blockquotes.

Nested blockquotes require tracking the depth of `>` characters and building a hierarchical structure of blockquote nodes. Lazy continuation means that once a blockquote starts, subsequent non-blank lines continue the blockquote even without `>` prefixes, until a blank line or new block type is encountered.

Implement blockquote parsing by counting the number of `>` characters at the start of each line to determine nesting depth. Maintain a stack of active blockquote levels, creating new nested blockquotes when depth increases and closing blockquotes when depth decreases. For lazy continuation, allow lines without `>` prefixes to continue the current blockquote level.

**⚠️ Pitfall: Horizontal Rule False Positives**

Horizontal rules are created by lines containing only three or more dashes, asterisks, or underscores, with optional whitespace. However, many parsers create false positives by not properly checking for other content on the line or by conflicting with Setext heading underlines.

A line like "--- some text" should not create a horizontal rule, and a line with three dashes under text should be checked for Setext heading interpretation before considering it a horizontal rule. Additionally, the characters must all be the same type—mixing dashes and asterisks should not create a rule.

Implement horizontal rule detection with a regex that anchors to the start and end of the line, ensures all characters are the same type, and requires at least three repetitions. Check for horizontal rules only after ruling out Setext headings to avoid conflicts.

#### ⚠️ Pitfall: Block Boundary Detection

Determining where one block ends and another begins is more complex than simply looking for blank lines. Different block types have different termination rules, and some blocks (like paragraphs) can be interrupted by other block types even without blank line separators.

For example, a paragraph can be immediately followed by a heading without a blank line separator, but a paragraph cannot be immediately followed by another paragraph. Similarly, fenced code blocks ignore all content until their closing fence, including lines that would normally start new block types.

Create a clear hierarchy of block interruption rules. Headings and horizontal rules can interrupt paragraphs immediately. Code blocks have their own continuation rules that override normal block detection. Use a state machine that tracks the current block type and consults type-specific continuation rules before attempting to start new blocks.

## Implementation Guidance

The block parser forms the foundation of the entire markdown parsing pipeline, so robust implementation is crucial for the system's overall reliability. The following guidance provides both infrastructure components and core parsing logic to support all the block types required in Milestone 1.

### Technology Recommendations

| Component | Simple Option | Advanced Option |
|---|---|---|
| Line Processing | Basic string splitting with manual parsing | Regular expression engine with compiled patterns |
| State Management | Simple enum states with switch statements | State pattern with polymorphic state objects |
| AST Construction | Direct node creation with manual tree building | Builder pattern with fluent interface |
| Lookahead Buffer | Single-element peek buffer | Buffered reader with arbitrary lookahead |
| Block Recognition | Hardcoded if-else chains for each block type | Strategy pattern with pluggable block recognizers |

For a beginner implementation, the simple options provide clear, debuggable code. The advanced options become valuable when extending the parser with custom block types or optimizing performance for large documents.

### Recommended File Structure

```
project-root/
  src/
    markdown_parser/
      __init__.py                 ← main parse_to_html() entry point
      preprocessor.py             ← line processing and normalization
      block_parser.py             ← core block parsing logic (implement this)
      inline_parser.py            ← inline element parsing (future milestone)
      ast_nodes.py                ← AST node classes and enums
      html_generator.py           ← HTML output generation (future milestone)
      utils.py                    ← shared utilities and constants
    tests/
      test_block_parser.py        ← block parsing unit tests
      test_integration.py         ← end-to-end parsing tests
      fixtures/                   ← markdown test files
        block_elements.md
        edge_cases.md
```

This structure separates concerns cleanly while keeping related functionality together. The `block_parser.py` module contains the core logic you'll implement, while infrastructure components provide support functionality.

Infrastructure Starter Code

File: `src/markdown_parser/ast_nodes.py`

PYTHON

```python
from enum import Enum
from typing import List, Optional, Dict, Any

class NodeType(Enum):
    DOCUMENT = "document"
    BLOCK = "block"
    INLINE = "inline"
    TEXT = "text"

class BlockType(Enum):
    PARAGRAPH = "paragraph"
    HEADING = "heading"
    CODE_BLOCK = "code_block"
    BLOCKQUOTE = "blockquote"
    HORIZONTAL_RULE = "horizontal_rule"
    LIST = "list"
    LIST_ITEM = "list_item"

class InlineType(Enum):
    STRONG = "strong"
    EMPHASIS = "emphasis"
    CODE = "code"
    LINK = "link"
    IMAGE = "image"
    LINE_BREAK = "line_break"

class ASTNode:
    """Base class for all AST nodes."""

    def __init__(self, node_type: NodeType, line_number: int = 0):
        self.node_type = node_type
        self.children: List['ASTNode'] = []
        self.parent: Optional['ASTNode'] = None
```

```
        self.line_number = line_number

    def add_child(self, child: 'ASTNode') -> None:
        """Add child node and set parent relationship."""
        child.parent = self
        self.children.append(child)

    def find_nodes_by_type(self, target_type) -> List['ASTNode']:
        """Find all descendant nodes of specified type."""
        results = []
        if self.node_type == target_type:
            results.append(self)
        for child in self.children:
            results.extend(child.find_nodes_by_type(target_type))
        return results

class BlockNode(ASTNode):
    """Represents a block-level element like paragraph, heading, or code block."""

    def __init__(self, block_type: BlockType, line_number: int = 0):
        super().__init__(NodeType.BLOCK, line_number)
        self.block_type = block_type
        self.inline_content: str = ""
        self.block_attributes: Dict[str, Any] = {}

    def set_content(self, content: str) -> None:
        """Set raw text content for block."""
        self.inline_content = content

    def set_attribute(self, key: str, value: Any) -> None:
        """Set element-specific attribute."""
```

```
        self.block_attributes[key] = value


    def get_attribute(self, key: str, default: Any = None) -> Any:
        """Get element-specific attribute."""
        return self.block_attributes.get(key, default)

class InlineNode(ASTNode):
    """Represents an inline element like emphasis, link, or code span."""


    def __init__(self, inline_type: InlineType, line_number: int = 0):
        super().__init__(NodeType.INLINE, line_number)
        self.inline_type = inline_type
        self.text_content: str = ""
        self.formatting_attributes: Dict[str, Any] = {}
```

File: `src/markdown_parser/preprocessor.py`

PYTHON

```python
import re

from typing import List

from dataclasses import dataclass


@dataclass

class LineInfo:

    """Information about a single line of markdown input."""

    content: str

    line_number: int

    is_blank: bool

    indent_level: int


class Preprocessor:

    """Handles input normalization and line-level analysis."""



    def normalize_line_endings(self, text: str) -> str:

        """Convert all line endings to Unix format."""

        return text.replace('\r\n', '\n').replace('\r', '\n')



    def split_into_lines(self, text: str) -> List[str]:

        """Split text preserving line numbers."""

        return text.split('\n')



    def is_blank_line(self, line: str) -> bool:

        """Check for whitespace-only lines."""

        return len(line.strip()) == 0



    def get_indentation_level(self, line: str) -> int:

        """Count leading spaces (tabs count as 4 spaces)."""

        indent = 0

        for char in line:

            if char == ' ':
```

```
                indent += 1
            elif char == '\t':
                indent += 4
            else:
                break
        return indent

    def process_input(self, text: str) -> List[LineInfo]:
        """Process raw markdown into analyzed lines."""
        normalized = self.normalize_line_endings(text)
        lines = self.split_into_lines(normalized)

        result = []
        for i, line in enumerate(lines):
            line_info = LineInfo(
                content=line,
                line_number=i + 1,
                is_blank=self.is_blank_line(line),
                indent_level=self.get_indentation_level(line)
            )
            result.append(line_info)

        return result
```

File: `src/markdown_parser/utils.py`

```python
import re

from typing import Dict


# HTML entity escaping table

HTML_ESCAPE_TABLE: Dict[str, str] = {

    '&': '&amp;',

    '<': '&lt;',

    '>': '&gt;',

    '"': '&quot;',

    "'": '&#x27;'

}


def escape_html(text: str) -> str:

    """Escape HTML special characters."""

    for char, entity in HTML_ESCAPE_TABLE.items():

        text = text.replace(char, entity)

    return text


def debug_print_ast(node, indent: int = 0) -> None:

    """Display AST structure for debugging."""

    prefix = "  " * indent

    node_info = f"{prefix}{node.node_type.value}"



    if hasattr(node, 'block_type'):

        node_info += f" ({node.block_type.value})"

    elif hasattr(node, 'inline_type'):

        node_info += f" ({node.inline_type.value})"



    if hasattr(node, 'inline_content') and node.inline_content:

        content_preview = node.inline_content[:50].replace('\n', '\\n')

        node_info += f': "{content_preview}"'
```

```
    print(node_info)


    for child in node.children:
        debug_print_ast(child, indent + 1)

# Compiled regex patterns for block recognition
ATX_HEADING_PATTERN = re.compile(r'^(#{1,6})\s+(.*?)(?:\s+#+)?$')
SETEXT_H1_UNDERLINE = re.compile(r'^=+\s*$')
SETEXT_H2_UNDERLINE = re.compile(r'^-+\s*$')
FENCED_CODE_START = re.compile(r'^```(\w*)\s*$')
FENCED_CODE_END = re.compile(r'^```\s*$')
HORIZONTAL_RULE_PATTERN = re.compile(r'^(\*{3,}|-{3,}|_{3,})\s*$')
BLOCKQUOTE_PATTERN = re.compile(r'^>\s?(.*)')
```

Core Logic Skeleton Code

File: `src/markdown_parser/block_parser.py`

PYTHON

```python
from typing import List, Optional, Iterator

from enum import Enum

import re


from .ast_nodes import ASTNode, BlockNode, BlockType, NodeType

from .preprocessor import LineInfo, Preprocessor

from .utils import (ATX_HEADING_PATTERN, SETEXT_H1_UNDERLINE, SETEXT_H2_UNDERLINE,

                    FENCED_CODE_START, FENCED_CODE_END, HORIZONTAL_RULE_PATTERN,

                    BLOCKQUOTE_PATTERN)


class BlockParserState(Enum):

    """Current state of the block parser."""

    LOOKING_FOR_BLOCK = "looking_for_block"

    IN_PARAGRAPH = "in_paragraph"

    IN_FENCED_CODE = "in_fenced_code"

    IN_INDENTED_CODE = "in_indented_code"

    IN_BLOCKQUOTE = "in_blockquote"


class BlockParser:

    """Parses block-level markdown elements into an AST."""



    def __init__(self):

        self.preprocessor = Preprocessor()

        self.state = BlockParserState.LOOKING_FOR_BLOCK

        self.current_block: Optional[BlockNode] = None

        self.root_document: Optional[ASTNode] = None

        self.line_buffer: List[LineInfo] = []

        self.current_line_index = 0



    def parse_blocks(self, markdown_text: str) -> ASTNode:

        """Main entry point for block parsing.
```

```
        Returns the root document node containing all parsed blocks.
        """
        # TODO 1: Use preprocessor to convert text into LineInfo objects
        # TODO 2: Create root document node to hold all blocks
        # TODO 3: Initialize parsing state and line buffer
        # TODO 4: Process all lines through the main parsing loop
        # TODO 5: Finalize any incomplete block at end of input
        # TODO 6: Return the completed document AST
        pass

    def process_line_sequence(self, lines: List[LineInfo]) -> None:
        """Process a sequence of lines through the block parser state machine.

        This is the main parsing loop that handles state transitions and
        delegates to specific block type handlers.
        """
        # TODO 1: Set up line iteration with lookahead capability
        # TODO 2: For each line, check if current block can continue
        # TODO 3: If current block cannot continue, finalize it
        # TODO 4: Attempt to start new block based on line characteristics
        # TODO 5: If no specific block type matches, default to paragraph
        # TODO 6: Handle end-of-input by finalizing any active block
        pass

    def peek_next_line(self) -> Optional[LineInfo]:
        """Look ahead at the next line without consuming it.

        Used for Setext heading detection and other lookahead needs.
        """
        # TODO 1: Check if there are more lines available
        # TODO 2: Return the next line without advancing current position
```

```
        # TODO 3: Return None if at end of input
        pass

    def can_current_block_continue(self, line: LineInfo) -> bool:
        """Check if the current block can accept this line as continuation.

        Different block types have different continuation rules.
        """
        # TODO 1: Check current parser state and block type
        # TODO 2: For paragraphs, continue unless blank line or new block starts
        # TODO 3: For fenced code, continue until closing fence
        # TODO 4: For indented code, continue while indentation is maintained
        # TODO 5: For blockquotes, handle both > prefixed and lazy continuation
        # TODO 6: Return False if current block should terminate
        pass

    def try_start_heading_block(self, line: LineInfo) -> bool:
        """Attempt to start an ATX or Setext heading block.

        Returns True if heading was successfully started.
        """
        # TODO 1: Try ATX heading pattern (# through ######)
        # TODO 2: Extract heading level and content from ATX match
        # TODO 3: Try Setext heading by looking ahead to next line
        # TODO 4: For Setext, check if next line is all = or - characters
        # TODO 5: Create BlockNode with HEADING type and appropriate attributes
        # TODO 6: Set heading level (1-6) and content in block attributes
        pass

    def try_start_code_block(self, line: LineInfo) -> bool:
        """Attempt to start a fenced or indented code block.
```

```
        Returns True if code block was successfully started.
        """
        # TODO 1: Check for fenced code block start (triple backticks)
        # TODO 2: Extract language hint if present after opening fence
        # TODO 3: Set parser state to IN_FENCED_CODE
        # TODO 4: Check for indented code block (4+ spaces, after blank line)
        # TODO 5: Verify indented code is preceded by block boundary
        # TODO 6: Create CODE_BLOCK node with appropriate attributes
        pass

    def try_start_blockquote(self, line: LineInfo) -> bool:
        """Attempt to start a blockquote block.

        Returns True if blockquote was successfully started.
        """
        # TODO 1: Check for > prefix at start of line
        # TODO 2: Extract content after > marker (handling optional space)
        # TODO 3: Handle nested blockquotes by counting > characters
        # TODO 4: Create BLOCKQUOTE node with nesting level attribute
        # TODO 5: Set parser state to IN_BLOCKQUOTE
        # TODO 6: Store first line of blockquote content
        pass

    def try_start_horizontal_rule(self, line: LineInfo) -> bool:
        """Attempt to create a horizontal rule block.

        Returns True if horizontal rule was successfully created.
        """
        # TODO 1: Check for horizontal rule pattern (3+ dashes, stars, or underscores)
        # TODO 2: Ensure all characters are the same type
```

```
        # TODO 3: Verify line contains only rule characters and whitespace
        # TODO 4: Create HORIZONTAL_RULE node (no content needed)
        # TODO 5: Add completed block to document immediately
        # TODO 6: Reset state to LOOKING_FOR_BLOCK
        pass

    def start_paragraph_block(self, line: LineInfo) -> None:
        """Start a new paragraph block with the given line.

        This is the default block type when no other pattern matches.
        """
        # TODO 1: Create new BlockNode with PARAGRAPH type
        # TODO 2: Set the line content as initial paragraph content
        # TODO 3: Set parser state to IN_PARAGRAPH
        # TODO 4: Store line number for error reporting
        pass

    def continue_current_block(self, line: LineInfo) -> None:
        """Add the current line to the active block being built.

        Handling varies by block type and parser state.
        """
        # TODO 1: Check current parser state to determine continuation behavior
        # TODO 2: For paragraphs, append line content with proper spacing
        # TODO 3: For fenced code, check for closing fence before appending
        # TODO 4: For indented code, strip required indentation before appending
        # TODO 5: For blockquotes, handle > prefix removal and lazy continuation
        # TODO 6: Update block content while preserving line structure
        pass

    def finalize_current_block(self) -> None:
```

```
        """Complete the current block and add it to the document.


        Performs any cleanup needed based on block type.
        """
        # TODO 1: Check if there is an active block to finalize
        # TODO 2: Trim trailing whitespace from block content if appropriate
        # TODO 3: Add completed block as child of root document
        # TODO 4: Clear current block reference and reset state
        # TODO 5: Handle any block-type-specific finalization (e.g., code language)
        pass
```

Milestone Checkpoint

After implementing the block parser, verify correct functionality with these specific tests:

Test Command: `python -m pytest tests/test_block_parser.py -v`

Expected Behavior:

- ATX headings from `#` through `######` create heading blocks with correct levels
- Setext headings with `===` and `---` underlines create h1 and h2 blocks
- Fenced code blocks preserve content exactly and capture language hints
- Indented code blocks handle 4-space indentation correctly
- Blockquotes handle both `>` prefixed and lazy continuation lines
- Paragraphs group consecutive lines and terminate at blank lines
- Horizontal rules recognize `---`, `***`, and `___` patterns

Manual Verification: Create a test file with this content:

```markdown
# Main Title

This is a paragraph with

multiple lines of text.
```

Subsection

```python
def hello():

    print("world")
```

> This is a blockquote with multiple lines.

Final paragraph.

```
Run your parser and verify the AST contains 6 blocks: heading, paragraph, heading, code_block, blockquote, horizontal_rule, and paragraph.

**Debugging Signs**:
- If headings aren't recognized, check ATX pattern regex and heading level extraction
- If paragraphs merge incorrectly, verify blank line handling in `can_current_block_continue`
- If code blocks lose content, check fence detection and content preservation logic
- If blockquotes don't handle continuation, verify lazy continuation rules
```

Inline Parser Design

> **Milestone(s):** Milestone 2: Inline Elements

The Text Surgery Mental Model

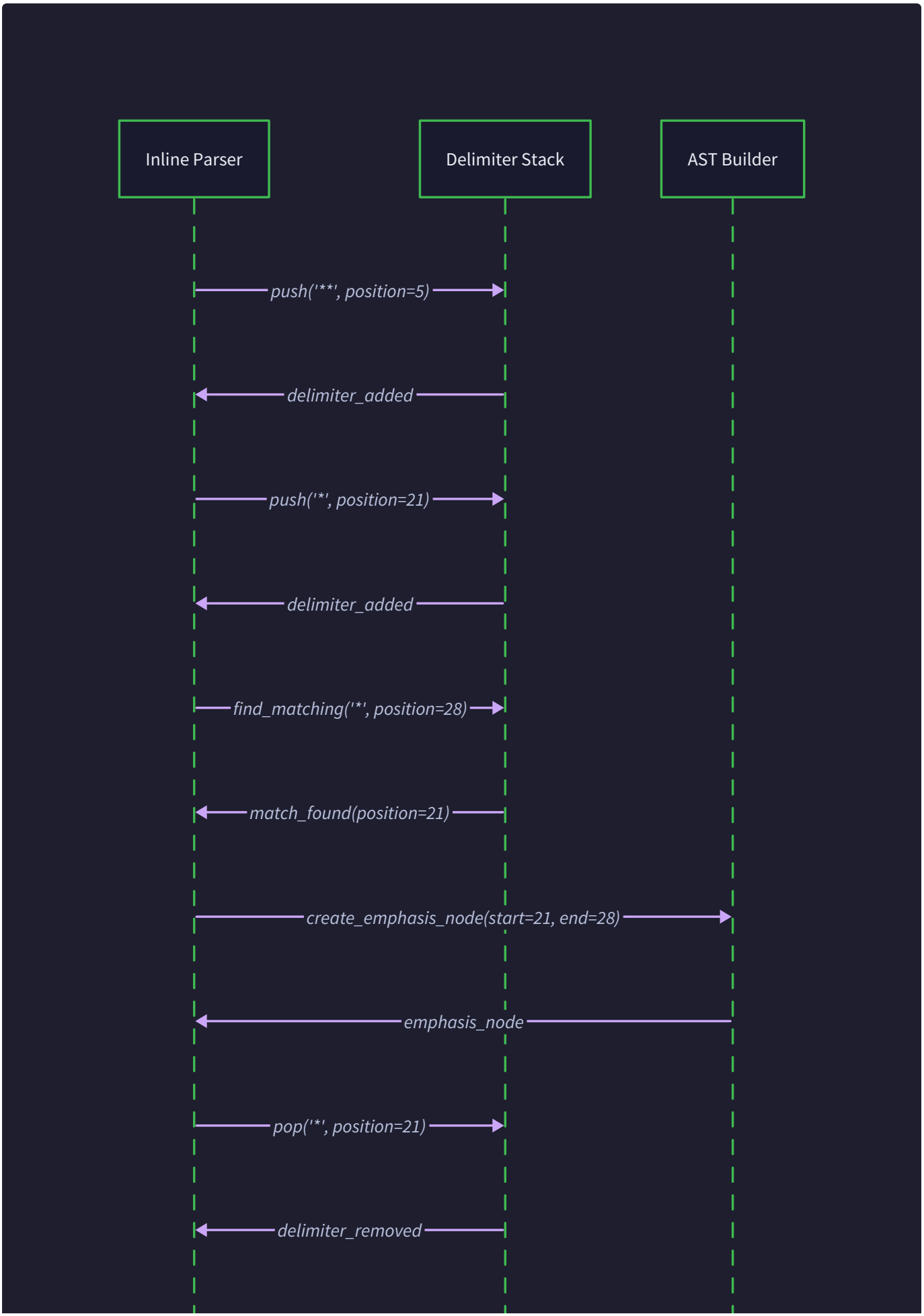
Think of inline parsing like performing precise surgery on a sentence. You have a paragraph of text that looks normal on the surface, but hidden within are formatting markers that need to be carefully extracted and transformed. Just as a surgeon must navigate around vital organs while removing a tumor, the inline parser must navigate around nested formatting while preserving the structure of the surrounding text.

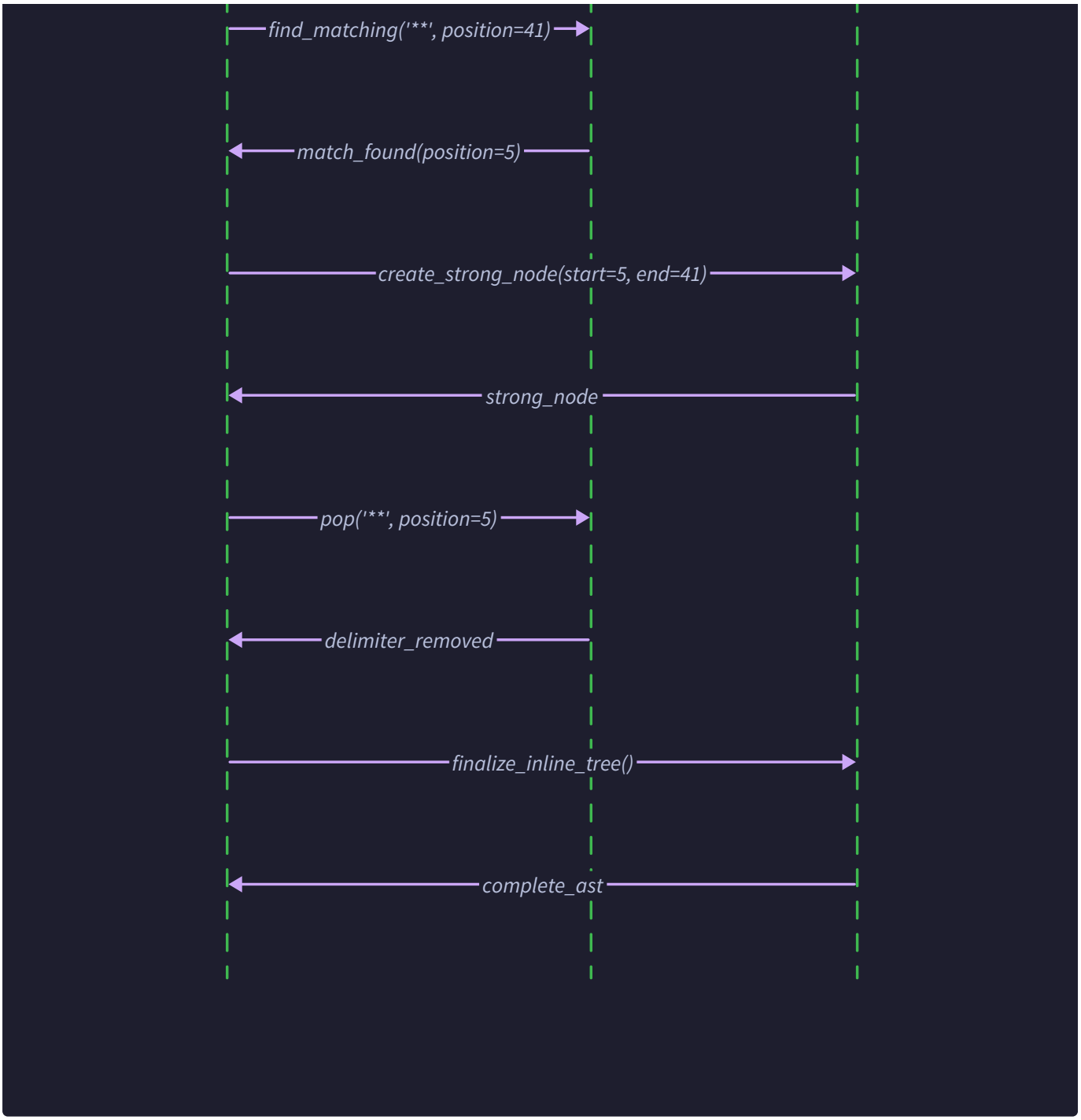
Consider the text: `This **bold text contains *nested italic* formatting** here.` The parser must recognize that the asterisks aren't just characters—they're surgical markers indicating where to make precise cuts. The double asterisks mark the boundaries of bold formatting, but within that region, single asterisks mark italic formatting. The parser must track the nesting depth, ensuring that when it closes the italic formatting, it doesn't accidentally close the bold formatting too early.

Unlike block parsing, which deals with entire lines and clear structural boundaries, inline parsing operates at the character level within continuous text. It's context-dependent parsing where the meaning of a character depends on what came before it and what surrounds it. An underscore in the middle of a word like `snake_case` should be treated as literal text, but underscores at word boundaries like `_emphasis_` should trigger formatting.

Inline Parsing Algorithm

The inline parser operates on text content that has already been extracted from block elements by the block parser. This two-phase approach allows the inline parser to focus purely on formatting markers without worrying about structural boundaries like paragraph breaks or code block delimiters.


Inline Parser
Delimiter Stack
AST Builder
push('**', position=5)
delimiter_added
push('*', position=21)
delimiter_added
find_matching('*', position=28)
match_found(position=21)
create_emphasis_node(start=21, end=28)
emphasis_node
pop('*', position=21)
delimiter_removed



The core algorithm uses a **delimiter stack approach** combined with **left-to-right scanning**. This handles the fundamental challenge of nested formatting where delimiters must be matched correctly even when they overlap or contain each other.

Primary Algorithm Steps

The inline parsing algorithm processes text through several coordinated phases:

1. **Character-by-character scanning**: The parser maintains a current position index and examines each character in the input text. This allows it to detect delimiter sequences that might span multiple characters, such as `**` for strong emphasis or `![` for image syntax.

2. **Delimiter detection and classification**: When the parser encounters a potential delimiter character (asterisk, underscore, backtick, square bracket), it must determine whether this character should be treated as a formatting marker or literal text. This involves checking the surrounding context, including whitespace, word boundaries, and escape sequences.

3. **Delimiter stack management**: Valid formatting delimiters are pushed onto a stack structure that tracks their position, type, and nesting level. The stack enables proper matching of opening and closing delimiters even when they're separated by other nested formatting.
4. **Content extraction**: Between matched delimiters, the parser recursively processes the contained text to handle nested formatting. This creates a tree structure where parent formatting elements contain child formatting elements.
5. **AST node creation**: Successfully matched delimiter pairs result in the creation of `InlineNode` instances with the appropriate `InlineType` values. These nodes are inserted into the AST with their text content and any formatting attributes.
6. **Escape sequence processing**: Throughout scanning, the parser must handle escape sequences where a backslash makes the following character literal. This prevents formatting markers from being processed when they're explicitly escaped by the author.

The algorithm handles several challenging cases that make inline parsing complex:

Emphasis delimiter precedence: When both asterisks and underscores are present, or when single and double delimiters compete, the parser must apply CommonMark precedence rules. Longer delimiter sequences (like `**`) generally take precedence over shorter ones (like `*`), and delimiters that are closer together are matched before those that are farther apart.

Intraword underscore handling: Underscores within words like `snake_case_variable` must not trigger emphasis formatting, while underscores at word boundaries should. The parser detects word boundaries by examining adjacent characters for alphanumeric content and Unicode word character properties.

Link and image parsing: These elements use complex syntax like `[link text](URL "title")` that requires careful parsing of multiple components. The parser must handle nested square brackets in link text, optional title attributes in quotes, and URL validation.

Inline code span precedence: Code spans delimited by backticks have special precedence—nothing inside them should be processed for other formatting. The parser must recognize code span boundaries first and treat their content as literal text for subsequent parsing phases.

Delimiter Stack Data Structures

The delimiter stack uses specialized data structures to track formatting state:

Field Name	Type	Description
`delimiter_type`	`str`	The delimiter character(s): '*', '**', '_', '__', '`', '[', '!['
`position`	`int`	Character index in the source text where delimiter was found
`can_open`	`bool`	Whether this delimiter can open formatting based on surrounding context
`can_close`	`bool`	Whether this delimiter can close formatting based on surrounding context
`is_active`	`bool`	Whether this delimiter is still available for matching
`delimiter_length`	`int`	Length of the delimiter sequence (1 for `*`, 2 for `**`)
`original_stack_position`	`int`	Position in delimiter stack for tracking nesting relationships

The stack enables several critical operations:

Push delimiter: When a potential opening delimiter is encountered, it's pushed onto the stack with context information about whether it can legitimately open formatting based on the surrounding whitespace and characters.

Match and pop: When a potential closing delimiter is found, the parser searches backward through the stack to find a matching opening delimiter of the same type. Intermediate delimiters may be deactivated if they interfere with the match.

Precedence resolution: When multiple valid matches are possible, the parser applies CommonMark precedence rules to determine which delimiters should be paired together.

Inline Parser Architecture Decisions

The inline parser design requires several critical architectural decisions that significantly impact both correctness and performance.

Decision: Delimiter Stack vs Regular Expression Matching

- **Context**: Inline formatting can be nested arbitrarily deep and must handle complex precedence rules between different delimiter types
- **Options Considered**:
  - Pure regex approach with complex patterns for each formatting type
  - Delimiter stack with explicit delimiter matching logic
  - Recursive descent parser with separate functions for each inline element type
- **Decision**: Delimiter stack with explicit matching logic
- **Rationale**: Regular expressions cannot handle arbitrary nesting depths due to their finite state nature, and complex regex patterns become unmaintainable. Recursive descent works but struggles with delimiter precedence when multiple formatting types interact. The delimiter stack approach directly models CommonMark's emphasis algorithm and handles both nesting and precedence correctly.
- **Consequences**: More complex implementation than regex but handles all edge cases correctly. Stack management adds some performance overhead but enables proper handling of malformed input.

Option	Pros	Cons	Chosen?
Regex Only	Simple implementation, fast for basic cases	Cannot handle nested formatting, complex patterns unmaintainable	No
Delimiter Stack	Handles nesting correctly, follows CommonMark spec exactly	More complex implementation, requires careful state management	**Yes**
Recursive Descent	Natural tree structure, easy to extend	Struggles with delimiter precedence, harder to handle malformed input	No

Decision: Two-Phase vs Single-Phase Inline Processing

- **Context**: Inline elements like code spans should take precedence over emphasis formatting, but links can contain emphasis in their text
- **Options Considered**:
  - Single pass processing all inline elements simultaneously
  - Two-phase approach: first pass for code spans and links, second pass for emphasis
  - Multi-phase approach with separate pass for each element type
- **Decision**: Two-phase approach with code spans processed first
- **Rationale**: Code spans must be processed before emphasis because nothing inside a code span should be formatted. Links need special handling because they can contain emphasis in their link text. Two phases balance correctness with implementation complexity.
- **Consequences**: Requires multiple passes over text but ensures correct precedence. Code spans properly mask their content from further processing.

Decision: Escape Sequence Processing Timing

- **Context**: Backslash escape sequences can appear anywhere and must prevent following characters from being interpreted as formatting
- **Options Considered**:
  - Process escapes during initial scanning before delimiter detection
  - Process escapes during delimiter matching after detection
  - Process escapes during HTML generation as final step
- **Decision**: Process escapes during initial scanning
- **Rationale**: Escaped characters should never be considered as potential delimiters, so escape processing must happen first. This prevents escaped asterisks from being pushed onto the delimiter stack at all.
- **Consequences**: Simplifies delimiter detection logic but requires careful handling of backslashes that aren't valid escape sequences.

Emphasis Delimiter Precedence Rules

The CommonMark specification defines complex precedence rules for emphasis delimiters that the parser must implement correctly:

Precedence Level	Rule	Example	Result
1	Longer delimiter sequences take precedence	`***text***`	`<strong><em>text</em></strong>`
2	Closer delimiter pairs are matched first	`*a **b* c**`	`<em>a **b</em> c**` (malformed)
3	Left-to-right processing for equal precedence	`*a* *b*`	`<em>a</em> <em>b</em>`
4	Intraword underscores are disabled	`snake_case_var`	No formatting applied
5	Asterisks work anywhere	`a*b*c`	`a<em>b</em>c`

These precedence rules require the delimiter stack to implement sophisticated matching logic that goes beyond simple stack operations.

Context-Dependent Delimiter Rules

The parser must implement context-dependent rules for determining when characters can function as opening or closing delimiters:

Context Check	Applies To	Rule	Example Valid	Example Invalid
Left-flanking	Opening delimiters	Not preceded by Unicode whitespace, not followed by punctuation unless preceded by whitespace/punctuation	`*emphasis*`	`* not emphasis`
Right-flanking	Closing delimiters	Not followed by Unicode whitespace, not preceded by punctuation unless followed by whitespace/punctuation	`*emphasis*`	`not emphasis *`
Intraword underscore	Underscores only	Cannot open/close if flanked by alphanumeric characters	`_start_`	`snake_case`
Code span priority	All delimiters	Cannot open/close inside code spans	`` `*literal*` ``	N/A

Inline Parsing Common Pitfalls

Inline parsing presents several challenging edge cases that frequently trip up implementers. Understanding these pitfalls helps avoid subtle bugs that can be difficult to debug.

⚠️ Pitfall: Underscore Intraword Handling

Many implementations incorrectly handle underscores within words, either formatting when they shouldn't or failing to format when they should. The CommonMark specification has specific rules: underscores cannot open or close emphasis if they're flanked by alphanumeric characters.

Consider `snake_case_variable` versus `_emphasis_word`. In the first case, both underscores are flanked by letters, so no emphasis should be applied. In the second case, the first underscore is preceded by whitespace (can open) and the second is followed by whitespace (can close), so emphasis should be applied.

The fix requires implementing proper flanking detection that checks the Unicode character classes of adjacent characters, not just simple whitespace detection. The parser must examine the character before the underscore and after to determine if they're alphanumeric.

⚠️ Pitfall: Delimiter Matching Precedence Errors

Implementers often use greedy matching that doesn't follow CommonMark precedence rules, leading to incorrect parsing of overlapping emphasis. For example, `***bold italic***` should parse as `<strong><em>bold italic</em></strong>` (double asterisk matched first, then single), not `<em><strong>bold italic</strong></em>`.

The issue occurs when parsers match the first opening delimiter they find with the first closing delimiter, rather than applying proper precedence rules. The CommonMark algorithm specifically handles this by processing longer delimiter sequences first and implementing complex matching logic.

The fix requires implementing the full CommonMark emphasis algorithm with proper precedence handling, not just simple stack matching.

⚠️ Pitfall: Escape Sequence Processing Order

Processing escape sequences at the wrong time leads to either double-escaping or failure to escape properly. If escapes are processed during HTML generation, then `\*` might be treated as a delimiter during parsing and incorrectly formatted.

The correct approach processes escapes during the initial scanning phase, converting `\*` to a literal asterisk before any delimiter detection occurs. This ensures that escaped characters are never considered as formatting markers.

⚠️ Pitfall: Code Span Precedence Violations

Many parsers incorrectly process emphasis formatting inside code spans, violating the rule that code spans take precedence over all other inline formatting. Text like `` `*literal asterisks*` `` should render with literal asterisks, not emphasis.

This happens when parsers don't implement proper two-phase processing or when they detect code spans incorrectly. The fix requires processing code spans first and marking their content as literal text that should be skipped during emphasis processing.

⚠️ Pitfall: Link Parsing Bracket Matching

Link syntax parsing often fails on nested brackets or complex URLs. Text like `[link with [nested] brackets](URL)` requires careful bracket counting to determine where the link text ends and the URL begins.

Simple regex patterns fail on this input because they can't handle the nested structure. The fix requires implementing proper bracket counting that tracks nesting depth and handles escape sequences within the link text.

⚠️ Pitfall: Malformed Input Recovery

Many parsers fail catastrophically on malformed input like unmatched delimiters or incomplete link syntax. Input like `**unmatched emphasis` should render with literal asterisks, not cause a parser error.

The correct approach implements graceful degradation where unmatched delimiters are left as literal text. This requires careful cleanup of the delimiter stack when processing completes, converting any unmatched delimiters back to literal characters.

Delimiter Stack State Management Issues

Several common issues arise from improper delimiter stack management:

Issue	Symptom	Cause	Fix
Stack overflow on deeply nested input	Parser crashes or hangs	No depth limit on delimiter nesting	Implement maximum nesting depth limit
Memory leak on malformed input	Memory usage grows unbounded	Unmatched delimiters never removed from stack	Clean up unmatched delimiters at text boundaries
Incorrect precedence handling	Wrong emphasis nesting in output	Matching first available delimiter instead of highest precedence	Implement proper precedence search in stack
Performance degradation on long text	Parsing becomes very slow	O(n²) behavior from excessive stack scanning	Optimize delimiter matching with better data structures

Context Detection Edge Cases

Context-dependent parsing creates several edge cases that are easy to get wrong:

Input	Expected Result	Common Wrong Result	Issue
`a_b_c`	`a_b_c` (no formatting)	`a<em>b</em>c`	Intraword underscore rule not implemented
`_a_b_`	`<em>a_b</em>`	`_a<em>b</em>_`	First underscore not recognized as opener
`**a*b**`	`<strong>a*b</strong>`	`**a<em>b</em>*`	Inner asterisk shouldn't close outer emphasis
`*a**b*c**`	`<em>a**b</em>c**`	Various incorrect parsings	Complex precedence interaction

Implementation Guidance

The inline parser implementation requires careful attention to character-level processing and state management. The following guidance provides both infrastructure components and core parsing logic to help implement a correct inline parser.

Technology Recommendations

Component	Simple Option	Advanced Option
Regex Engine	Built-in `re` module	Third-party `regex` module with Unicode support
Character Classification	Manual ASCII checks	`unicodedata` module for proper Unicode handling
Stack Implementation	Python `list` with append/pop	Custom stack class with additional metadata
String Building	String concatenation	`io.StringIO` for efficient building
Delimiter Detection	Character-by-character scanning	Compiled regex patterns for common delimiters

Recommended File Structure

```
markdown_parser/
  parsers/
    inline_parser.py           ← main inline parsing logic
    delimiter_stack.py         ← delimiter stack implementation
    inline_elements.py         ← specific element parsers (links, emphasis)
    escape_processor.py        ← escape sequence handling
  ast/
    inline_nodes.py           ← InlineNode and InlineType definitions
  utils/
    text_utils.py             ← character classification and flanking detection
  tests/
    test_inline_parser.py     ← comprehensive inline parsing tests
    test_emphasis_precedence.py ← specific emphasis precedence tests
```

PYTHON

Infrastructure Starter Code

Complete Delimiter Stack Implementation (delimiter_stack.py):

PYTHON

```python
from dataclasses import dataclass

from typing import List, Optional, Tuple

from enum import Enum


class DelimiterType(Enum):

    ASTERISK = "asterisk"

    UNDERSCORE = "underscore"

    BACKTICK = "backtick"

    LEFT_BRACKET = "left_bracket"

    EXCLAMATION_BRACKET = "exclamation_bracket"


@dataclass

class Delimiter:

    delimiter_type: DelimiterType

    char: str

    position: int

    length: int

    can_open: bool

    can_close: bool

    is_active: bool = True

    original_length: int = None



    def __post_init__(self):

        if self.original_length is None:

            self.original_length = self.length


class DelimiterStack:

    def __init__(self):

        self.stack: List[Delimiter] = []



    def push(self, delimiter: Delimiter) -> None:

        """Add a delimiter to the stack."""
```

```
        self.stack.append(delimiter)

    def find_matching_opener(self, closer: Delimiter) -> Optional[Tuple[int, Delimiter]]:
        """Find the most recent matching opener for the given closer."""
        for i in range(len(self.stack) - 1, -1, -1):
            opener = self.stack[i]
            if (opener.is_active and
                opener.can_open and
                opener.delimiter_type == closer.delimiter_type and
                opener.char == closer.char):
                return i, opener
        return None

    def deactivate_between(self, start_idx: int, end_idx: int) -> None:
        """Deactivate delimiters between two positions."""
        for i in range(start_idx + 1, end_idx):
            if i < len(self.stack):
                self.stack[i].is_active = False

    def clear_to_position(self, position: int) -> None:
        """Remove delimiters up to a specific text position."""
        self.stack = [d for d in self.stack if d.position >= position]

    def get_unmatched_delimiters(self) -> List[Delimiter]:
        """Get all unmatched delimiters for literal text conversion."""
        return [d for d in self.stack if d.is_active]

def detect_flanking_properties(text: str, pos: int, delimiter_len: int) -> Tuple[bool, bool]:
    """Detect if delimiter can open/close based on flanking rules."""
    import unicodedata
```

```
    # Get characters before and after delimiter
    before_char = text[pos - 1] if pos > 0 else ' '
    after_char = text[pos + delimiter_len] if pos + delimiter_len < len(text) else ' '

    # Unicode character classification
    before_whitespace = before_char.isspace()
    after_whitespace = after_char.isspace()
    before_punctuation = unicodedata.category(before_char).startswith('P')
    after_punctuation = unicodedata.category(after_char).startswith('P')

    # Left-flanking: not followed by whitespace, and either
    # not followed by punctuation or preceded by whitespace/punctuation
    left_flanking = (not after_whitespace and
                    (not after_punctuation or before_whitespace or before_punctuation))

    # Right-flanking: not preceded by whitespace, and either
    # not preceded by punctuation or followed by whitespace/punctuation
    right_flanking = (not before_whitespace and
                     (not before_punctuation or after_whitespace or after_punctuation))

    return left_flanking, right_flanking

def can_open_emphasis(char: str, left_flanking: bool, right_flanking: bool,
                      before_char: str, after_char: str) -> bool:
    """Determine if delimiter can open emphasis."""
    if char == '*':
        return left_flanking
    elif char == '_':
        return left_flanking and (not right_flanking or not after_char.isalnum())
    return False

def can_close_emphasis(char: str, left_flanking: bool, right_flanking: bool,
```

```
                    before_char: str, after_char: str) -> bool:
    """Determine if delimiter can close emphasis."""
    if char == '*':
        return right_flanking
    elif char == '_':
        return right_flanking and (not left_flanking or not before_char.isalnum())
    return False
```

Complete Escape Processor (escape_processor.py):

```python
import re

from typing import List, Tuple


# Characters that can be escaped in CommonMark

ESCAPABLE_CHARS = r'!"#$%&\'()*+,\-./:;<=>?@[\\\]^_`{|}~'


class EscapeProcessor:

    def __init__(self):

        self.escape_pattern = re.compile(r'\\(.)')



    def process_escapes(self, text: str) -> Tuple[str, List[int]]:

        """

        Process escape sequences and return processed text with escape positions.



        Returns:

            Tuple of (processed_text, list of positions that were escaped)

        """

        result = []

        escaped_positions = []

        i = 0



        while i < len(text):

            if text[i] == '\\' and i + 1 < len(text):

                next_char = text[i + 1]

                if next_char in ESCAPABLE_CHARS:

                    # Valid escape sequence

                    result.append(next_char)

                    escaped_positions.append(len(result) - 1)

                    i += 2

                else:

                    # Invalid escape, keep backslash

                    result.append(text[i])
```

```
                i += 1
            else:
                result.append(text[i])
                i += 1

        return ''.join(result), escaped_positions

    def is_escaped(self, position: int, escaped_positions: List[int]) -> bool:
        """Check if a character at given position was escaped."""
        return position in escaped_positions
```

Core Logic Skeleton Code

Main Inline Parser (inline_parser.py):

PYTHON

```python
from typing import List, Optional, Dict, Any
from .delimiter_stack import DelimiterStack, Delimiter, DelimiterType
from .escape_processor import EscapeProcessor
from ..ast.inline_nodes import InlineNode, InlineType
from ..ast.ast_nodes import ASTNode

class InlineParser:
    def __init__(self):
        self.delimiter_stack = DelimiterStack()
        self.escape_processor = EscapeProcessor()

    def parse_inline_elements(self, text: str) -> List[InlineNode]:
        """
        Main entry point for parsing inline elements from text.

        Returns list of InlineNode objects representing the parsed content.
        """
        # TODO 1: Process escape sequences first to handle \* and other escapes
        # TODO 2: First pass - process code spans (highest precedence)
        # TODO 3: Second pass - process links and images
        # TODO 4: Third pass - process emphasis (asterisks and underscores)
        # TODO 5: Convert any remaining text to text nodes
        # TODO 6: Return list of inline nodes
        pass

    def _process_code_spans(self, text: str, escaped_positions: List[int]) -> List[InlineNode]:
        """
        Process inline code spans delimited by backticks.
        Code spans have highest precedence and mask their content.
        """
        # TODO 1: Scan for backtick sequences, ignoring escaped backticks
```

```
        # TODO 2: Match opening and closing backtick sequences of same length
        # TODO 3: Extract code content and normalize whitespace per CommonMark rules
        # TODO 4: Create InlineNode with InlineType.CODE
        # TODO 5: Mark code span regions as processed for subsequent passes
        pass

    def _process_emphasis(self, text: str, escaped_positions: List[int],
                          processed_regions: List[Tuple[int, int]]) -> List[InlineNode]:
        """
        Process emphasis (bold and italic) using delimiter matching algorithm.
        """
        self.delimiter_stack = DelimiterStack()  # Reset for each text segment

        # TODO 1: Scan character by character for emphasis delimiters
        # TODO 2: For each potential delimiter, check flanking properties
        # TODO 3: Push valid opening delimiters onto stack
        # TODO 4: For closing delimiters, find matching opener with precedence rules
        # TODO 5: Create InlineNode for matched delimiter pairs
        # TODO 6: Handle unmatched delimiters as literal text
        # TODO 7: Recursively process content within emphasis nodes
        pass

    def _scan_emphasis_delimiters(self, text: str, pos: int) -> Optional[Delimiter]:
        """
        Scan for emphasis delimiters at current position.
        Returns Delimiter object if valid delimiter found, None otherwise.
        """
        # TODO 1: Check for asterisk or underscore at current position
        # TODO 2: Count consecutive delimiter characters (*, **, ***, etc.)
        # TODO 3: Determine flanking properties using detect_flanking_properties
        # TODO 4: Apply emphasis opening/closing rules based on delimiter type
```

```
        # TODO 5: Return Delimiter object with all properties set
        pass


    def _process_links_and_images(self, text: str, escaped_positions: List[int],
                                  processed_regions: List[Tuple[int, int]]) -> List[InlineNode]:
        """
        Process link and image syntax: [text](url) and ![alt](url)
        """
        # TODO 1: Scan for opening brackets [ and image syntax ![
        # TODO 2: Handle nested brackets in link text by counting bracket depth
        # TODO 3: Look for closing bracket followed by opening parenthesis
        # TODO 4: Parse URL and optional title in parentheses
        # TODO 5: Create InlineNode with InlineType.LINK or InlineType.IMAGE
        # TODO 6: Recursively process inline formatting in link text
        # TODO 7: Handle malformed links as literal text
        pass


    def _match_emphasis_delimiters(self, closer: Delimiter) -> Optional[Tuple[int, Delimiter]]:
        """
        Find matching opener for emphasis closer using CommonMark precedence.
        """
        # TODO 1: Search delimiter stack backward for matching opener
        # TODO 2: Apply precedence rules - longer sequences first
        # TODO 3: Check delimiter type compatibility (asterisk vs underscore)
        # TODO 4: Ensure opener can open and closer can close
        # TODO 5: Return stack index and opener delimiter if found
        pass


    def _create_emphasis_node(self, opener: Delimiter, closer: Delimiter,
                              inner_content: str) -> InlineNode:
        """
```

```
        Create emphasis node based on delimiter length and type.
        """
        # TODO 1: Determine emphasis type based on delimiter length (1=em, 2=strong)
        # TODO 2: Create InlineNode with appropriate InlineType
        # TODO 3: Recursively parse inner content for nested formatting
        # TODO 4: Set text_content and formatting_attributes
        # TODO 5: Return completed InlineNode
        pass
```

Link and Image Parser (inline_elements.py):

```python
import re

from typing import Optional, Tuple, Dict, Any

from ..ast.inline_nodes import InlineNode, InlineType


class LinkImageParser:

    def __init__(self):

        # Regex patterns for URL and title parsing

        self.url_pattern = re.compile(r'^[^\s<>]*(?:\s+"[^"]*")?')

        self.title_pattern = re.compile(r'\s+"([^"]*)"$')



    def parse_link_or_image(self, text: str, start_pos: int,

                            is_image: bool) -> Optional[Tuple[InlineNode, int]]:

        """

        Parse link or image syntax starting at given position.



        Returns:

            Tuple of (InlineNode, end_position) if successful, None if malformed

        """

        # TODO 1: Skip opening marker ([ or ![) and find link text

        # TODO 2: Handle nested brackets in link text with bracket counting

        # TODO 3: Verify closing bracket followed by opening parenthesis

        # TODO 4: Parse URL and optional title from parentheses

        # TODO 5: Create InlineNode with InlineType.LINK or InlineType.IMAGE

        # TODO 6: Return node and position after closing parenthesis

        pass



    def _find_link_text_end(self, text: str, start_pos: int) -> Optional[int]:

        """

        Find the end of link text, handling nested brackets.

        """

        # TODO 1: Track bracket nesting depth starting from 1
```

```
        # TODO 2: Scan character by character, ignoring escaped brackets
        # TODO 3: Increment depth on '[', decrement on ']'
        # TODO 4: Return position when depth reaches 0
        # TODO 5: Return None if no matching bracket found
        pass


    def _parse_url_and_title(self, text: str) -> Tuple[str, Optional[str]]:
        """
        Parse URL and optional title from parentheses content.
        """
        # TODO 1: Apply regex to extract URL portion
        # TODO 2: Check for optional title in quotes after URL
        # TODO 3: Handle angle-bracket enclosed URLs
        # TODO 4: Return tuple of (url, title) where title may be None
        pass
```

Milestone Checkpoints

After completing emphasis parsing:

- Test command: `python -m pytest tests/test_inline_parser.py::test_emphasis -v`
- Expected behavior: `**bold**` becomes `<strong>bold</strong>`, `*italic*` becomes `<em>italic</em>`
- Manual verification: Create test file with `**bold *nested* text**` and verify nested structure
- Debug check: Print delimiter stack contents to verify proper delimiter matching

After completing link parsing:

- Test command: `python -m pytest tests/test_inline_parser.py::test_links -v`
- Expected behavior: `[text](url)` becomes `<a href="url">text</a>`
- Manual verification: Test nested brackets `[link [with] brackets](url)` parses correctly
- Debug check: Verify bracket counting logic handles nested cases

After completing code spans:

- Test command: `python -m pytest tests/test_inline_parser.py::test_code_spans -v`
- Expected behavior: `` `code` `` becomes `<code>code</code>`, content is literal
- Manual verification: `` `*not emphasis*` `` should not format the asterisks
- Debug check: Confirm code spans are processed before emphasis in pipeline

Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
Emphasis not nested correctly	Wrong delimiter precedence	Print delimiter stack during matching	Implement proper precedence rules in `_match_emphasis_delimiters`
Underscores in words get formatted	Missing intraword underscore check	Test with `snake_case` input	Add alphanumeric flanking check to `can_open_emphasis`
Links with brackets don't parse	Bracket counting bug	Log bracket depth during parsing	Fix bracket nesting logic in `_find_link_text_end`
Code spans don't mask emphasis	Processing order wrong	Check if code spans processed first	Move code span processing to first pass
Escaped characters still format	Escape processing timing	Verify escaped positions list	Process escapes before delimiter detection
Memory usage grows on long text	Delimiter stack not cleaned	Monitor stack size during parsing	Clear stack between text segments

List Parser Design

> **Milestone(s):** Milestone 3: Lists

The Document Hierarchy Mental Model

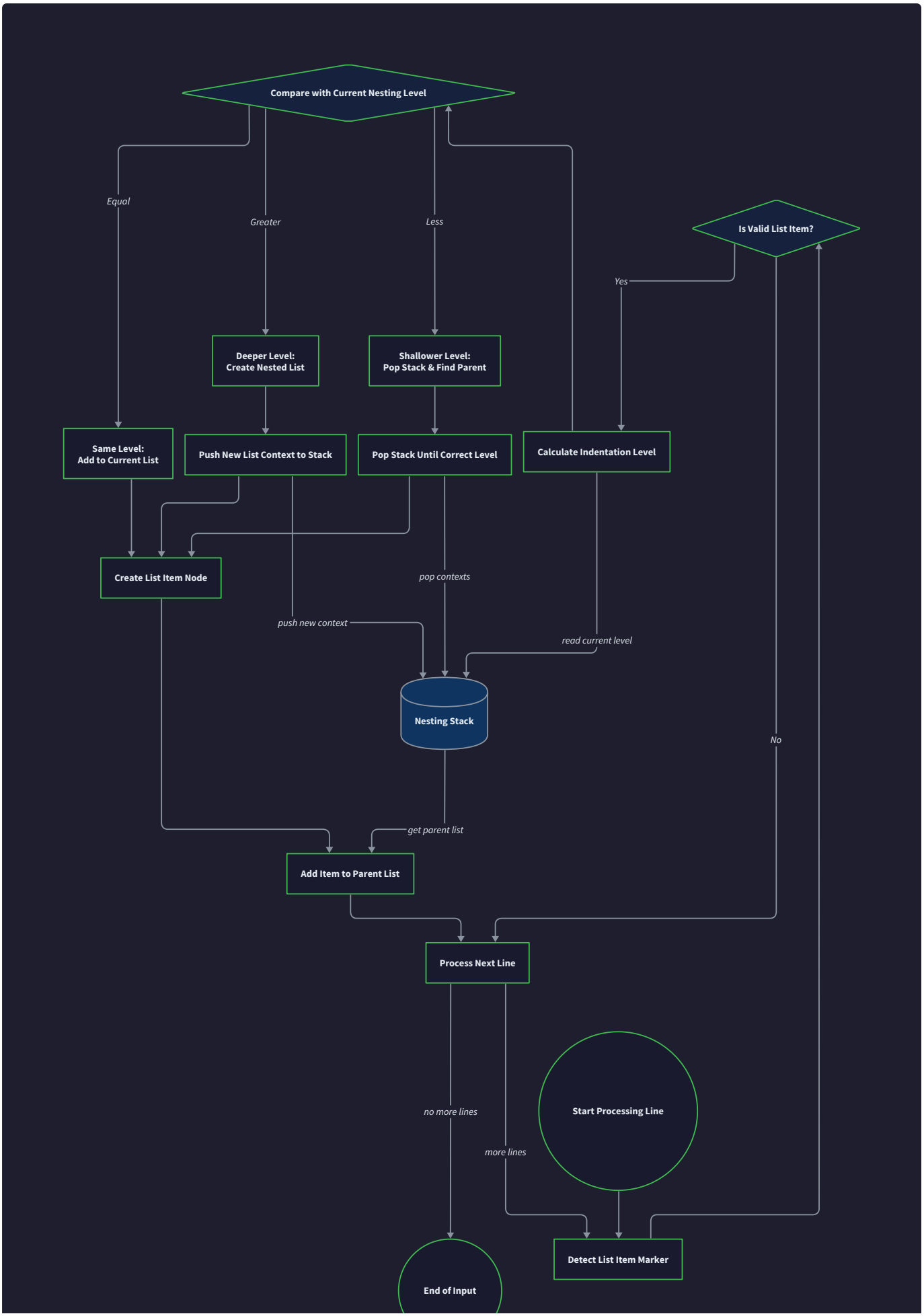
Think of list parsing like organizing a company's org chart from a messy pile of business cards. Each business card has a person's name and shows their reporting level through indentation - the CEO has no indentation, VPs have one level, directors have two levels, and so on. Your job is to reconstruct the proper hierarchical tree structure by carefully tracking indentation levels and understanding who reports to whom.

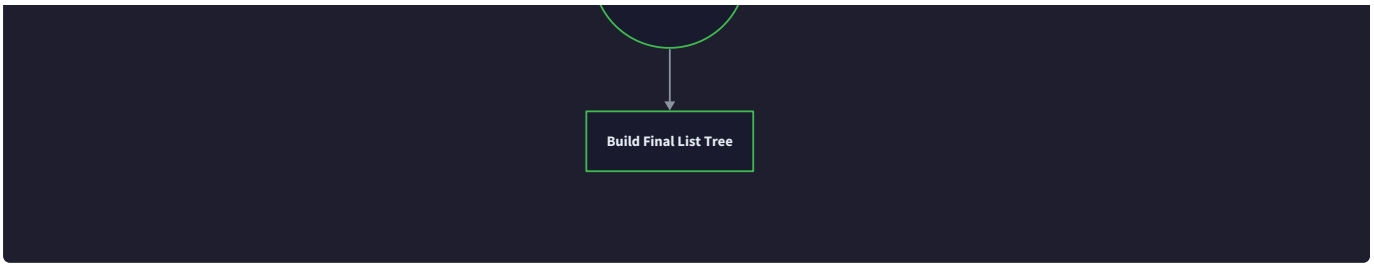
Just like how a person can't report to someone at a lower level in the hierarchy, markdown list items follow strict nesting rules based on indentation. A list item indented 4 spaces can only be a child of an item indented 0-3 spaces, never a sibling or parent. The challenge is that unlike a clean org chart, the "business cards" (markdown lines) arrive in sequential order, and you must build the hierarchy incrementally while handling edge cases like missing intermediate levels or inconsistent indentation.

The fundamental insight is that **list parsing is inherently recursive and stateful** - each list item's position in the hierarchy depends not just on its own indentation, but on the context of all previous items and the current nesting stack. This requires maintaining a sophisticated understanding of the document's hierarchical state as parsing progresses.

List Parsing Algorithm

The list parsing algorithm operates as a **stateful hierarchy builder** that maintains a stack of active list contexts while processing lines sequentially. The core challenge is correctly interpreting indentation levels and determining when to create new nested lists versus continuing existing ones.


Compare with Current Nesting Level
Equal
Greater
Less
Is Valid List Item?
Yes
Deeper Level:
Create Nested List
Shallower Level:
Pop Stack & Find Parent
Same Level:
Add to Current List
Push New List Context to Stack
Pop Stack Until Correct Level
Calculate Indentation Level
Create List Item Node
pop contexts
push new context
read current level
Nesting Stack
No
get parent list
Add Item to Parent List
Process Next Line
no more lines
Start Processing Line
more lines
End of Input
Detect List Item Marker



Phase 1: List Item Detection and Classification

The algorithm begins by identifying potential list items through marker detection. This phase involves examining each line to determine if it represents a list item and, if so, what type of list it belongs to.

1. **Line Analysis**: Extract the line's leading whitespace to calculate the base indentation level using `get_indentation_level()`. Remove leading whitespace to examine the content that follows.
2. **Marker Detection**: Check if the line begins with a valid list marker. For unordered lists, look for dash (`-`), asterisk (`*`), or plus (`+`) followed by at least one space. For ordered lists, look for a sequence of 1-9 digits followed by either a period (`.`) or closing parenthesis (`)`) and at least one space.
3. **Marker Validation**: Verify that the marker is properly formatted. The marker must be followed by at least one space or tab, or be at the end of the line (creating an empty list item). Calculate the marker's consumed width to determine the content indentation.
4. **Content Indentation Calculation**: Determine the indentation level for list item content by adding the base indentation to the marker width plus following spaces. This becomes the reference indentation for continuation lines.
5. **List Type Determination**: Based on the marker type, classify the item as either ordered or unordered. For ordered lists, extract the starting number, though CommonMark specifies that the actual numbering in output starts from the first item's number.

Phase 2: Hierarchy Management and Nesting

Once a list item is detected, the algorithm must determine its position in the document hierarchy and manage the nesting stack appropriately.

1. **Context Stack Evaluation**: Compare the current item's indentation with the indentation levels stored in the active context stack. The context stack maintains information about currently open lists and their indentation levels.
2. **Stack Unwinding**: If the current item's indentation is less than or equal to any level in the stack, unwind the stack by finalizing and closing list contexts until reaching an appropriate parent level. This handles cases where the nesting level decreases.
3. **Sibling vs Child Determination**: If the indentation matches an existing level in the stack, create a sibling item at that level. If the indentation is greater than all existing levels but within valid nesting bounds, create a new child list.
4. **New List Creation**: When starting a new nested level, create a new `BlockNode` with `block_type` set to `LIST`. Set the list type attribute (ordered/unordered) and add it as a child to the current list item. Push this new context onto the stack.
5. **List Item Creation**: Create a `BlockNode` with `block_type` set to `LIST_ITEM`. Set the raw content as the text following the marker. Add this item to the current list context and mark it as the active item for potential continuation lines.

Phase 3: Content Processing and Continuation

List items can contain multiple lines of content, including nested paragraphs, code blocks, and other structural elements. The algorithm must handle these continuation patterns correctly.

1. **Continuation Line Detection**: For each subsequent line, determine if it continues the current list item, starts a new list item, or terminates the list structure. This requires checking indentation against the current list item's content indentation level.
2. **Content Type Classification**: Continuation lines may be simple paragraph text, blank lines (which affect tight vs loose list determination), or the beginning of nested block elements like code blocks or nested lists.
3. **Lazy Continuation Handling**: CommonMark allows "lazy continuation" where continuation lines may have less indentation than required, as long as they don't start a new block element. Implement this by checking if under-indented lines could plausibly continue the current list item.
4. **Nested Block Processing**: When continuation lines indicate nested block elements (like indented code blocks or nested blockquotes), delegate parsing to the appropriate block parser while maintaining the list context.
5. **Tight vs Loose List Determination**: Track whether blank lines appear between list items or within list items. Lists containing blank lines become "loose" lists, which affects HTML output (loose lists wrap content in paragraph tags).

List Parser Architecture Decisions

The list parser's design involves several critical architectural decisions that affect both correctness and implementation complexity. Each decision involves trade-offs between parsing accuracy, implementation simplicity, and performance.

> **Decision: Indentation Tracking Strategy**
>
> - **Context**: List nesting requires precise indentation tracking, but markdown allows various indentation styles and lazy continuation. The parser needs to handle both strict CommonMark compliance and graceful degradation for malformed input.
> - **Options Considered**: Character-level position tracking, space-count normalization, tab-expansion with configurable width
> - **Decision**: Normalize tabs to spaces using 4-space tab stops, then track indentation as space counts with a tolerance mechanism for minor variations
> - **Rationale**: Tab normalization eliminates ambiguity in mixed-indentation documents. Space counting provides precise nesting level determination. Tolerance allows graceful handling of minor indentation inconsistencies common in hand-written markdown.
> - **Consequences**: Enables reliable nesting detection and CommonMark compliance. Requires upfront tab normalization. May mask some user errors that strict parsers would reject.

Indentation Strategy	Pros	Cons
Character Position Tracking	Exact position preservation, handles mixed tabs/spaces	Complex logic, fragile with copy/paste
Space Count Normalization	Simple integer arithmetic, robust	Loses original formatting intent
Tab Expansion + Space Count	CommonMark compliant, predictable	Requires configuration, upfront processing

Decision: List Context Stack Management

- **Context**: Nested lists require maintaining multiple active contexts simultaneously. The parser must track which lists are open, their types, indentation levels, and current state for proper nesting and termination.
- **Options Considered**: Recursive descent parsing, explicit context stack, state machine with embedded stack
- **Decision**: Explicit context stack with list metadata including indentation level, list type, tight/loose status, and parent references
- **Rationale**: Explicit stack provides clear visibility into nesting state for debugging. Metadata tracking enables correct CommonMark compliance for tight/loose lists. Parent references enable proper AST construction.
- **Consequences**: Simplifies nesting logic and makes state transitions explicit. Requires careful stack management and cleanup. Enables comprehensive error reporting with context.

Context Management	Pros	Cons
Recursive Descent	Natural nesting representation, clean code	Stack overflow risk, complex backtracking
Explicit Context Stack	Clear state visibility, controlled memory usage	Manual stack management, more bookkeeping
State Machine + Stack	Systematic state handling, good error recovery	Complex state explosion, harder to debug

Decision: Tight vs Loose List Handling

- **Context**: CommonMark distinguishes between tight lists (no blank lines) and loose lists (containing blank lines), which affects HTML output. Tight lists don't wrap content in paragraph tags, while loose lists do.
- **Options Considered**: Post-processing analysis, incremental tracking during parsing, defer decision to HTML generator
- **Decision**: Incremental tracking during parsing with list-level and item-level blank line flags stored in block attributes
- **Rationale**: Incremental tracking avoids expensive post-processing passes. List-level flags enable correct handling of mixed scenarios. Early determination enables better error messages and debugging.
- **Consequences**: Enables correct CommonMark-compliant output generation. Requires careful blank line tracking. Complicates list item processing logic slightly.

Tight/Loose Strategy	Pros	Cons
Post-processing Analysis	Clean separation of concerns, simple parsing	Extra traversal pass, complex analysis logic
Incremental Tracking	Single-pass efficiency, immediate feedback	Distributed logic, more state tracking
HTML Generator Decision	Deferred complexity, flexible output control	Generator complexity, limited error reporting

List Type Consistency and Mixed Lists

The parser must handle scenarios where list markers change within what appears to be a single list. CommonMark specifies that different marker types or ordered/unordered mixing creates separate lists.

Decision: Mixed List Marker Handling

- **Context**: Users sometimes mix different list markers (`-`, `*`, `+`) or combine ordered and unordered items, expecting them to be part of the same list. CommonMark treats different markers as separate lists.
- **Options Considered**: Strict CommonMark compliance (separate lists), marker normalization (treat all as same), user configuration option
- **Decision**: Strict CommonMark compliance with clear error reporting when marker types change
- **Rationale**: Ensures predictable output consistent with other CommonMark parsers. Encourages consistent markdown authoring. Avoids ambiguity in complex nested scenarios.
- **Consequences**: May surprise users expecting mixed markers to work. Enables reliable round-trip parsing. Simplifies nesting logic significantly.

Continuation Line Processing Strategy

List items can contain multiple paragraphs, code blocks, and other complex content. The parser must distinguish between content that belongs to the current list item versus content that terminates the list.

Decision: Continuation Line Classification

- **Context**: Lines following list items may be continuation content, lazy continuation (under-indented but still belonging to the item), new list items, or content that terminates the list entirely.
- **Options Considered**: Strict indentation matching, lazy continuation support, lookahead-based classification
- **Decision**: Support lazy continuation with lookahead to distinguish from new block elements, using content indentation thresholds
- **Rationale**: Lazy continuation is part of CommonMark spec and matches user expectations. Lookahead prevents false positives where under-indented lines start new blocks. Thresholds provide clear rules for classification.
- **Consequences**: Enables natural authoring patterns users expect. Requires more complex line classification logic. May accept some ambiguous input that strict parsers reject.

List Parsing Common Pitfalls

List parsing involves several subtle edge cases that frequently cause implementation errors. Understanding these pitfalls helps build robust parsers that handle real-world markdown correctly.

⚠️ Pitfall: Incorrect Indentation Calculation

Many implementations incorrectly calculate the indentation required for list item continuation by only considering the marker width, ignoring the base indentation of the list item itself. This leads to incorrect nesting decisions and malformed output.

For example, consider this markdown:

```markdown
- Item with base indentation

  Continuation line
```

The continuation line requires 4 spaces total: 2 for the base indentation plus 2 for the marker and space. Implementations that only check for 2 spaces of indentation will incorrectly treat the continuation as a new paragraph outside the list.

Fix: Always calculate continuation indentation as `base_indentation + marker_width + min_space_requirement`. Store this value when creating each list item and use it consistently for all continuation line checks.

⚠️ Pitfall: Mixed List Type Confusion

Implementations often incorrectly handle scenarios where ordered and unordered list markers appear at the same indentation level, either by treating them as the same list or by not properly closing the previous list before starting the new one.

```markdown
1. First ordered item

- First unordered item

2. Second ordered item?
```

The unordered item should start a new list, and the final line should start yet another ordered list, not continue the first one.

Fix: Track the list type (ordered/unordered) and marker style for each list context. When a different type or marker style is encountered at the same indentation level, finalize the current list and start a new one. Store list type in the context stack along with indentation information.

⚠️ Pitfall: Loose List Detection Errors

Incorrect handling of blank lines leads to wrong tight/loose list classification, causing HTML output that doesn't match CommonMark specification. The most common error is not distinguishing between blank lines that separate list items versus blank lines within list items.

```markdown
1. First item


2. Second item



   With continuation paragraph
```

This should be a loose list because of the blank line between items, and the continuation paragraph should be wrapped in `<p>` tags.

Fix: Track blank lines at both the list level and item level. Set the loose flag when blank lines appear between list items or when any individual item contains blank lines. Propagate loose status to parent lists when nested loose lists are encountered.

⚠️ Pitfall: Lazy Continuation Boundary Errors

Implementations often incorrectly handle lazy continuation by either being too permissive (allowing continuation that should start new blocks) or too restrictive (requiring strict indentation that CommonMark doesn't mandate).

```markdown
- List item

This should continue the item

  - But this starts a nested list
```

The middle line is lazy continuation, but the final line starts a nested list despite being more indented than the continuation.

Fix: Implement lookahead to check if under-indented lines could start new block elements. Use the `can_current_block_continue()` method to test if a line belongs to the current list item before accepting it as lazy continuation.

⚠️ Pitfall: Nested List Context Cleanup

Failing to properly clean up list contexts when nesting levels decrease leads to memory leaks and incorrect parent-child relationships in the AST. This often manifests as lists that never properly close or items that appear in the wrong parent list.

```markdown
- Level 1
  - Level 2
    - Level 3
- Back to level 1
```

The final item should close all nested contexts and return to the top level.

Fix: Implement proper stack unwinding when indentation decreases. Use the `finalize_current_block()` method to close contexts and ensure proper parent-child relationships. Maintain parent references to enable correct AST construction.

⚠️ Pitfall: Empty List Item Handling

Empty list items (markers followed by blank lines or end of input) often break parsing logic that assumes list items always have content. This leads to null pointer exceptions or malformed AST nodes.

```markdown
- First item
-
- Third item
```

The middle item is empty but valid according to CommonMark.

Fix: Create list item nodes even for empty items, using empty string content. Handle empty content gracefully in continuation processing and HTML generation. Set appropriate attributes to distinguish truly empty items from items with whitespace-only content.

⚠️ Pitfall: Ordered List Starting Number Confusion

Many implementations either ignore ordered list starting numbers entirely or incorrectly propagate them to nested lists. CommonMark specifies that ordered lists should start from the number specified in the first item, but nested lists always start from 1.

```markdown
3. Starting from three
4. Next item
   1. Nested starts from one
   2. Continues normally
5. Back to main sequence
```

Fix: Store the starting number from the first item in each ordered list context using `set_attribute('start_number', num)`. Reset to 1 for nested ordered lists. Pass starting numbers to the HTML generator for proper `<ol start="3">` attribute generation.

Implementation Guidance

The list parser implementation requires careful coordination between indentation tracking, context management, and content processing. The following guidance provides practical patterns and utilities for building a robust list parser.

Technology Recommendations

Component	Simple Option	Advanced Option
Indentation Tracking	String splitting + len() counting	Regular expressions with capture groups
Context Stack	Python list with dict contexts	Custom ListContext class with methods
Marker Detection	String startswith() + manual parsing	Compiled regex patterns with named groups
Content Processing	Line-by-line state machine	Recursive descent with backtracking
Tight/Loose Tracking	Boolean flags in context dict	Enumerated state with validation rules

Recommended File Structure

```
project-root/
  src/
    markdown_parser/
      __init__.py              ← main parser interface
      preprocessor.py          ← line normalization and analysis
      block_parser/
        __init__.py            ← block parser interface
        block_parser.py        ← main block parsing logic
        list_parser.py         ← list-specific parsing logic (this component)
        list_context.py        ← list context stack management
        list_utils.py          ← indentation and marker utilities
      ast_nodes.py             ← AST node definitions
      common/
        line_info.py           ← LineInfo and related utilities
        patterns.py            ← regex patterns and constants
```

Infrastructure Starter Code (Complete and Ready to Use)

list_utils.py - Indentation and marker detection utilities:

```python
import re

from typing import Optional, Tuple, NamedTuple

from enum import Enum


class MarkerType(Enum):

    UNORDERED_DASH = "-"

    UNORDERED_ASTERISK = "*"

    UNORDERED_PLUS = "+"

    ORDERED_PERIOD = "."

    ORDERED_PAREN = ")"


class ListMarkerInfo(NamedTuple):

    marker_type: MarkerType

    is_ordered: bool

    marker_width: int

    content_indent: int

    start_number: Optional[int] = None


# Regex patterns for list marker detection

UNORDERED_MARKER_PATTERN = re.compile(r'^( {0,3})([-*+])( +)')

ORDERED_MARKER_PATTERN = re.compile(r'^( {0,3})([1-9][0-9]{0,8})([.])( +)')


def normalize_line_endings(text: str) -> str:

    """Convert all line endings to Unix format."""

    return text.replace('\r\n', '\n').replace('\r', '\n')


def expand_tabs_to_spaces(text: str, tab_width: int = 4) -> str:

    """Expand tabs to spaces using specified tab width."""

    result = []

    column = 0

    for char in text:

        if char == '\t':

            spaces_to_add = tab_width - (column % tab_width)
```

```
            result.append(' ' * spaces_to_add)
            column += spaces_to_add
        else:
            result.append(char)
            if char == '\n':
                column = 0
            else:
                column += 1
    return ''.join(result)

def get_indentation_level(line: str) -> int:
    """Count leading spaces in a line."""
    count = 0
    for char in line:
        if char == ' ':
            count += 1
        else:
            break
    return count

def detect_list_marker(line: str) -> Optional[ListMarkerInfo]:
    """Detect and parse list marker information from a line."""
    # Try unordered markers first
    match = UNORDERED_MARKER_PATTERN.match(line)
    if match:
        base_indent = len(match.group(1))
        marker_char = match.group(2)
        spaces_after = len(match.group(3))


        marker_type_map = {
            '-': MarkerType.UNORDERED_DASH,
            '*': MarkerType.UNORDERED_ASTERISK,
```

```
            '+': MarkerType.UNORDERED_PLUS
        }

        marker_width = 1 + spaces_after
        content_indent = base_indent + marker_width

        return ListMarkerInfo(
            marker_type=marker_type_map[marker_char],
            is_ordered=False,
            marker_width=marker_width,
            content_indent=content_indent
        )

    # Try ordered markers
    match = ORDERED_MARKER_PATTERN.match(line)
    if match:
        base_indent = len(match.group(1))
        number_str = match.group(2)
        delimiter = match.group(3)
        spaces_after = len(match.group(4))

        marker_type = MarkerType.ORDERED_PERIOD if delimiter == '.' else MarkerType.ORDERED_PAREN
        marker_width = len(number_str) + 1 + spaces_after
        content_indent = base_indent + marker_width

        return ListMarkerInfo(
            marker_type=marker_type,
            is_ordered=True,
            marker_width=marker_width,
            content_indent=content_indent,
            start_number=int(number_str)
```

```
        )


    return None

def is_blank_line(line: str) -> bool:
    """Check if line contains only whitespace."""
    return len(line.strip()) == 0

def can_lazy_continue(line: str, required_indent: int) -> bool:
    """Check if a line can be lazy continuation of a list item."""
    if is_blank_line(line):
        return True


    actual_indent = get_indentation_level(line)


    # Allow lazy continuation if not starting a new block element
    if actual_indent < required_indent:
        # Check if line could start a new list item
        if detect_list_marker(line) is not None:
            return False


        # Check for other block elements (simplified check)
        stripped = line.lstrip()
        if (stripped.startswith('#') or
            stripped.startswith('>') or
            stripped.startswith('```') or
            stripped.startswith('---')):
            return False


        return True
```

```
    return True
```

list_context.py - Context stack management:

PYTHON

```python
from typing import List, Optional, Dict, Any

from dataclasses import dataclass, field

from enum import Enum

from ..ast_nodes import ASTNode, BlockNode, BlockType

from .list_utils import MarkerType


class ListState(Enum):

    TIGHT = "tight"

    LOOSE = "loose"

    UNDETERMINED = "undetermined"


@dataclass

class ListContext:

    """Context information for an active list."""

    list_node: BlockNode

    marker_type: MarkerType

    base_indentation: int

    content_indentation: int

    is_ordered: bool

    state: ListState = ListState.UNDETERMINED

    start_number: Optional[int] = None

    item_count: int = 0

    current_item: Optional[BlockNode] = None

    has_blank_lines: bool = False



    def add_item(self, item_node: BlockNode) -> None:

        """Add a new list item to this list."""

        self.list_node.add_child(item_node)

        self.current_item = item_node

        self.item_count += 1



    def mark_loose(self) -> None:
```

```
        """Mark this list as loose (containing blank lines)."""
        self.state = ListState.LOOSE
        self.list_node.set_attribute('tight', False)

    def finalize(self) -> None:
        """Finalize list state when parsing is complete."""
        if self.state == ListState.UNDETERMINED:
            self.state = ListState.TIGHT
            self.list_node.set_attribute('tight', True)

        if self.is_ordered and self.start_number is not None:
            self.list_node.set_attribute('start', self.start_number)

class ListContextStack:
    """Manages the stack of active list contexts."""

    def __init__(self):
        self.contexts: List[ListContext] = []

    def current_context(self) -> Optional[ListContext]:
        """Get the current (top) list context."""
        return self.contexts[-1] if self.contexts else None

    def push_context(self, context: ListContext) -> None:
        """Push a new list context onto the stack."""
        self.contexts.append(context)

    def pop_context(self) -> Optional[ListContext]:
        """Pop and finalize the top context."""
        if self.contexts:
            context = self.contexts.pop()
```

```
            context.finalize()
            return context
        return None

    def unwind_to_indentation(self, target_indent: int) -> List[ListContext]:
        """Unwind contexts until reaching target indentation level."""
        finalized = []
        while (self.contexts and
               self.contexts[-1].base_indentation >= target_indent):
            finalized.append(self.pop_context())
        return finalized

    def find_context_for_indentation(self, indent: int) -> Optional[ListContext]:
        """Find the context that should contain an item at given indentation."""
        for context in reversed(self.contexts):
            if context.base_indentation <= indent:
                return context
        return None

    def mark_all_loose(self) -> None:
        """Mark all contexts as loose (blank lines affect all levels)."""
        for context in self.contexts:
            context.mark_loose()

    def clear(self) -> List[ListContext]:
        """Clear all contexts and return them finalized."""
        finalized = []
        while self.contexts:
            finalized.append(self.pop_context())
        return finalized
```

Core Logic Skeleton Code

list_parser.py - Main list parsing logic:

PYTHON

```python
from typing import List, Optional, Iterator
from ..ast_nodes import ASTNode, BlockNode, BlockType, NodeType
from ..common.line_info import LineInfo
from .list_context import ListContextStack, ListContext, ListState
from .list_utils import (
    detect_list_marker, get_indentation_level, is_blank_line,
    can_lazy_continue, MarkerType, ListMarkerInfo
)

class ListParser:
    """Handles parsing of ordered and unordered lists with nesting."""

    def __init__(self):
        self.context_stack = ListContextStack()
        self.current_line_index = 0
        self.lines: List[LineInfo] = []

    def parse_list_sequence(self, lines: List[LineInfo], start_index: int) -> tuple[List[BlockNode], int]:
        """
        Parse a sequence of list items starting at the given index.
        Returns (list_nodes, next_index) where next_index is the first line
        that doesn't belong to any list.
        """
        # TODO 1: Initialize parsing state with lines and starting index
        # TODO 2: Process lines sequentially until no more list content
        # TODO 3: Handle context stack cleanup and finalization
        # TODO 4: Return completed list nodes and next line index
        # Hint: Use process_line_for_lists() for each line
        pass

    def process_line_for_lists(self, line: LineInfo) -> bool:
```

```
        """
        Process a single line in the context of list parsing.
        Returns True if the line was consumed by list parsing, False otherwise.
        """
        # TODO 1: Check if line is blank and handle loose list marking
        # TODO 2: Try to detect list marker using detect_list_marker()
        # TODO 3: If marker found, handle as new list item with handle_list_item()
        # TODO 4: If no marker, try continuation with try_list_continuation()
        # TODO 5: If neither works, finalize contexts and return False
        # Hint: Blank lines between items mark lists as loose
        pass

    def handle_list_item(self, line: LineInfo, marker_info: ListMarkerInfo) -> None:
        """
        Handle a line that contains a list marker.
        Creates appropriate list structures and adds the item.
        """
        # TODO 1: Calculate base indentation from line and marker info
        # TODO 2: Determine if this starts new list or continues existing
        # TODO 3: Unwind context stack if indentation decreased
        # TODO 4: Create new list context if needed with create_list_context()
        # TODO 5: Create list item node and add to current list
        # TODO 6: Extract item content after marker and set content
        # Hint: Use context_stack.find_context_for_indentation() to find parent
        pass

    def try_list_continuation(self, line: LineInfo) -> bool:
        """
        Try to continue the current list item with this line.
        Returns True if line was consumed as continuation.
        """
```

```
        # TODO 1: Check if any list contexts are active
        # TODO 2: Get current context and required indentation
        # TODO 3: Check if line meets continuation requirements
        # TODO 4: Use can_lazy_continue() to allow lazy continuation
        # TODO 5: Append line content to current item if valid continuation
        # TODO 6: Handle blank lines within items (mark loose if needed)
        # Hint: Continuation requires indentation >= context.content_indentation
        pass

    def create_list_context(self, marker_info: ListMarkerInfo, base_indent: int) -> ListContext:
        """
        Create a new list context for the given marker and indentation.
        """
        # TODO 1: Create BlockNode with LIST block_type
        # TODO 2: Set list attributes (ordered/unordered, marker type)
        # TODO 3: If ordered list, set start number from marker_info
        # TODO 4: Create ListContext with all required parameters
        # TODO 5: Add list node to parent context if nesting
        # TODO 6: Return the new context for pushing to stack
        # Hint: Check if this is a nested list and attach to current item
        pass

    def check_marker_compatibility(self, current_marker: MarkerType, new_marker: MarkerType) -> bool:
        """
        Check if a new marker is compatible with the current list type.
        Different markers or ordered/unordered mixing starts new lists.
        """
        # TODO 1: Extract ordered/unordered status from both markers
        # TODO 2: Return False if mixing ordered and unordered
        # TODO 3: For ordered lists, check delimiter compatibility (. vs ))
        # TODO 4: For unordered lists, allow any unordered marker mix
```

```
        # TODO 5: Return True if markers can be in same list
        # Hint: CommonMark allows mixing -, *, + but not ordered/unordered
        pass

    def extract_item_content(self, line: str, marker_info: ListMarkerInfo) -> str:
        """
        Extract the content portion of a list item line after the marker.
        """
        # TODO 1: Calculate total prefix length (indentation + marker + spaces)
        # TODO 2: Slice line content after the prefix
        # TODO 3: Handle case where line ends immediately after marker
        # TODO 4: Strip trailing whitespace but preserve internal spacing
        # TODO 5: Return cleaned content string
        # Hint: marker_info.content_indent tells you where content starts
        pass

    def finalize_all_contexts(self) -> List[BlockNode]:
        """
        Finalize all remaining list contexts and return root list nodes.
        """
        # TODO 1: Get all contexts from stack using context_stack.clear()
        # TODO 2: Finalize each context to set tight/loose attributes
        # TODO 3: Collect root list nodes (those not nested in other lists)
        # TODO 4: Return list of root nodes for adding to document
        # Hint: Root nodes are those whose parent is not a list item
        pass

    def handle_blank_line_in_lists(self) -> None:
        """
        Handle blank lines that appear within list structures.
        Marks lists as loose according to CommonMark rules.
```

```
        """
        # TODO 1: Check if we're currently in any list context
        # TODO 2: Mark current context as having blank lines
        # TODO 3: Apply loose marking rules (blank between items vs within items)
        # TODO 4: Use context_stack.mark_all_loose() if between items
        # Hint: Blank lines between items make all parent lists loose
        pass
```

Language-Specific Hints

Python Implementation Tips:

- Use `str.lstrip()` and `str.rstrip()` for whitespace handling, but be careful about internal spacing preservation
- The `re` module's named groups (`(?P<name>...)`) make marker parsing more readable
- List comprehensions can simplify context stack filtering: `[ctx for ctx in contexts if ctx.base_indentation < target]`
- Use `dataclasses.dataclass` for context objects to get automatic `__init__` and `__repr__` methods
- Python's `enumerate()` function helps track line numbers during processing
- Use `collections.deque` if you need efficient line lookahead with `peek_next_line()` functionality

Regex Pattern Optimization:

- Compile regex patterns once at module level rather than recreating them for each line
- Use raw strings (`r"..."`) for regex patterns to avoid double-escaping backslashes
- Named capture groups improve code readability: `r"^(?P<indent> {0,3})(?P<marker>[-*+])(?P<space> +)"`
- Consider using `re.VERBOSE` flag for complex patterns to allow comments and formatting

Milestone Checkpoint

After implementing the list parser component, verify correct functionality with these specific tests:

Test Command: `python -m pytest tests/test_list_parser.py -v`

Expected Behaviors to Verify:

1. **Simple List Parsing**:

```python
markdown = "- Item 1\n- Item 2\n- Item 3"
# Should produce ul with 3 li elements
```

2. **Nested List Structure**:

```python
markdown = "1. First\n   - Nested\n   - Items\n2. Second"
# Should produce ol containing li with nested ul
```

3. **Tight vs Loose Lists**:

```python
tight = "- Item 1\n- Item 2"  # No blank lines

loose = "- Item 1\n\n- Item 2"  # Blank line between

# Tight should have tight=True, loose should have tight=False
```

4. **Mixed Indentation Handling**:

```python
markdown = "  - Indented list\n    - Nested item\n  - Back to level 1"

# Should correctly track indentation levels and nesting
```

Manual Verification Steps:

1. Run `python -c "from markdown_parser import parse_to_html; print(parse_to_html('- Test\n - Nested'))"`
2. Check that nested lists produce proper `<ul><li>` structures with correct nesting
3. Verify that ordered lists use `<ol start="N">` when starting number is not 1
4. Confirm that loose lists wrap item content in `<p>` tags while tight lists don't

Common Issues and Diagnostics:

- **Lists not nesting**: Check indentation calculation in `get_indentation_level()` and context stack management
- **Wrong tight/loose detection**: Verify blank line tracking in `handle_blank_line_in_lists()`
- **Mixed list problems**: Check `check_marker_compatibility()` logic for ordered/unordered mixing
- **Content not continuing**: Debug `can_lazy_continue()` and continuation indentation requirements

Debug Output Helpers: Use `debug_print_ast()` to visualize the parsed AST structure and verify correct parent-child relationships in nested lists. The output should show clear hierarchical structure with proper list and list-item nesting.

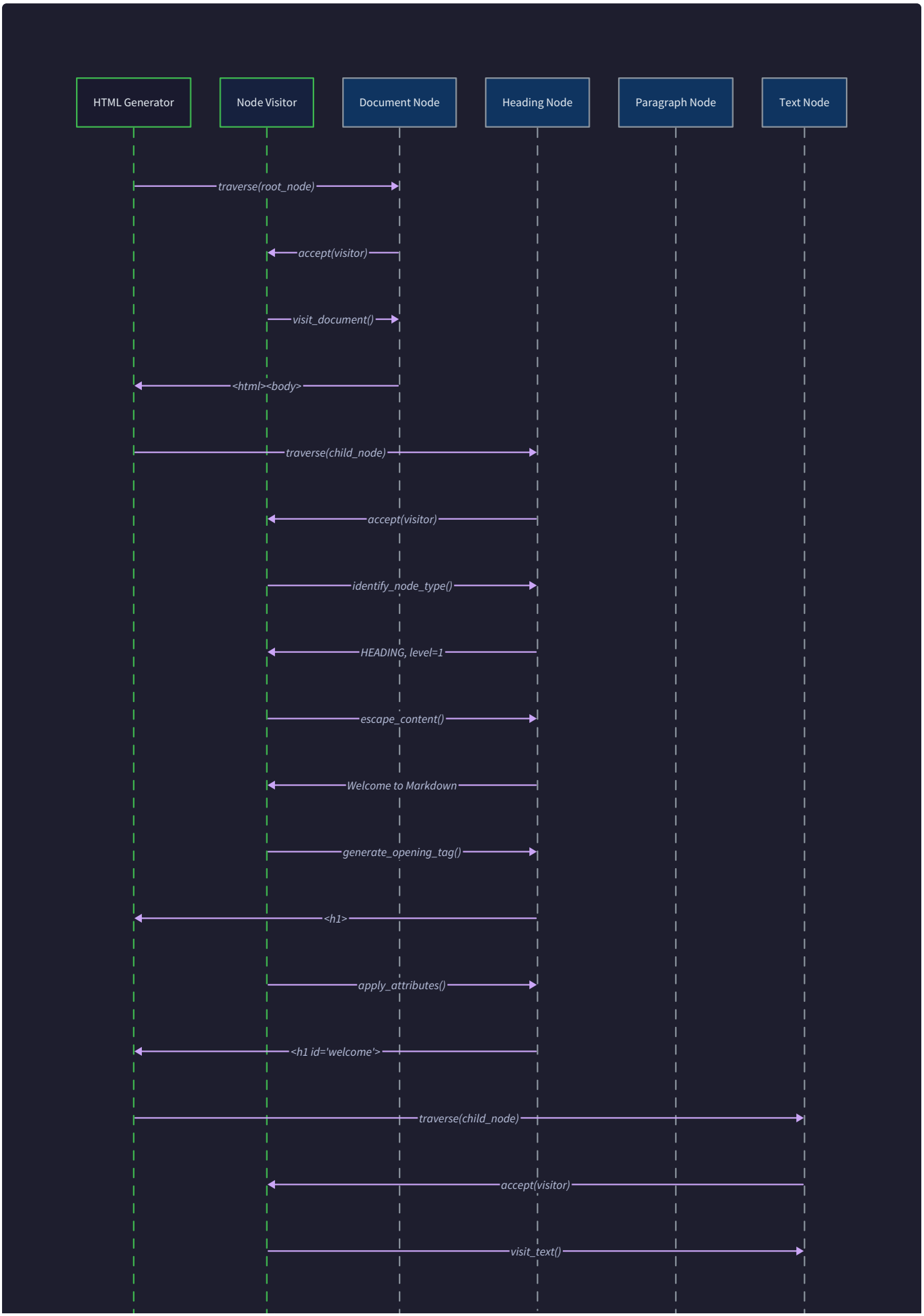
HTML Generator Design

> **Milestone(s):** Milestone 4: HTML Generation

The Document Assembly Mental Model

Think of HTML generation like assembling a piece of furniture from parsed instruction components. You have all the individual pieces (your AST nodes) with their specifications and relationships clearly defined, and now you need to transform them into the final assembled product (valid HTML) following precise construction rules. Just as furniture assembly requires the right tools (HTML escaping), the correct order of operations (depth-first traversal), and attention to detail (proper nesting and formatting), HTML generation requires systematic conversion of each AST component into its corresponding HTML representation while maintaining structural integrity.

The key insight is that HTML generation is fundamentally a **translation process** - you're converting from one structured representation (AST) to another (HTML markup) while preserving all the semantic meaning and relationships. This is different from parsing, which discovers structure from unstructured text. Here, the structure already exists and must be faithfully represented in a different format.


HTML Generator
Node Visitor
Document Node
Heading Node
Paragraph Node
Text Node
traverse(root_node)
accept(visitor)
visit_document()
<html><body>
traverse(child_node)
accept(visitor)
identify_node_type()
HEADING, level=1
escape_content()
Welcome to Markdown
generate_opening_tag()
<h1>
apply_attributes()
<h1 id='welcome'>
traverse(child_node)
accept(visitor)
visit_text()

HTML Generation Algorithm

The HTML generation process follows a systematic **depth-first tree traversal** pattern where each AST node is visited exactly once and converted to its corresponding HTML representation. This approach ensures that nested structures are properly handled and that parent-child relationships in the AST translate correctly to HTML element nesting.

The core algorithm operates through a **visitor pattern** where different node types trigger different rendering behaviors. This design allows for extensible rendering where new node types can be added without modifying the core traversal logic, and custom renderers can override specific element generation while preserving the overall structure.

Tree Traversal Algorithm Steps

The HTML generation follows these sequential steps for each node in the AST:

1. **Node Type Identification**: Examine the current node's `node_type` and `block_type` or `inline_type` to determine the appropriate HTML element to generate. This dispatch mechanism routes each node to its specialized rendering logic.
2. **Pre-processing Content Escaping**: Before generating any HTML tags, process the node's text content through the HTML escaping system. This step must occur before any inline parsing results are incorporated to avoid double-escaping scenarios.
3. **Opening Tag Generation**: Create the appropriate opening HTML tag based on the node type. For block elements like headings, this involves tags like `<h1>` through `<h6>`. For inline elements, this includes tags like `<strong>`, `<em>`, or `<code>`. Self-closing elements like `<hr>` and `<img>` require special handling.
4. **Attribute Application**: Apply any attributes stored in the node's `formatting_attributes` or `block_attributes` dictionaries. This includes `href` attributes for links, `src` and `alt` attributes for images, `class` attributes for code blocks with language specifications, and any custom attributes defined by extensions.
5. **Child Node Recursive Processing**: For nodes with children in their `children` list, recursively invoke the HTML generation algorithm on each child node in order. This step maintains the depth-first traversal pattern and ensures proper nesting of HTML elements.
6. **Content Integration**: Combine the processed child content with any direct text content from the current node. For leaf nodes, this involves directly including the escaped text content. For container nodes, this involves wrapping the child HTML with the current node's tags.

7. **Closing Tag Generation**: Generate the appropriate closing HTML tag to match the opening tag created in step 3. Self-closing elements skip this step. Proper tag matching is critical for valid HTML output.
8. **Output Formatting**: Apply consistent indentation and line breaks according to the configured pretty-printing rules. This step enhances human readability without affecting the semantic meaning of the generated HTML.

Node Type Rendering Mappings

The following table defines how each AST node type maps to HTML elements:

AST Node Type	HTML Element	Attributes	Special Handling
`DOCUMENT`	`html` with `head` / `body`	`lang` from document attributes	Wrapper template generation
`PARAGRAPH`	`p`	Standard block attributes	Tight list detection affects wrapping
`HEADING`	`h1` through `h6`	`id` from heading text slug	Level determined by heading depth
`CODE_BLOCK`	`pre` containing `code`	`class` for language hint	Preserve exact whitespace and line breaks
`BLOCKQUOTE`	`blockquote`	Citation attributes if present	Recursive processing for nested quotes
`HORIZONTAL_RULE`	`hr`	None (self-closing)	No content or children
`LIST`	`ul` or `ol`	`start` attribute for ordered lists	Tight vs loose list content handling
`LIST_ITEM`	`li`	Task list attributes for checkboxes	Paragraph wrapping based on list tightness
`STRONG`	`strong`	None	Inline element with nested content
`EMPHASIS`	`em`	None	Inline element with nested content
`CODE`	`code`	None	Preserve inner whitespace exactly
`LINK`	`a`	`href`, optional `title`	URL escaping and validation
`IMAGE`	`img`	`src`, `alt`, optional `title`	Self-closing with required attributes
`LINE_BREAK`	`br`	None (self-closing)	Hard line break representation
`TEXT`	Raw text content	None	HTML entity escaping only

Content Processing Pipeline

Each node's content undergoes a multi-stage processing pipeline before becoming final HTML:

1. **Raw Content Extraction**: Extract the `text_content` or `inline_content` from the node, handling None values gracefully by treating them as empty strings.
2. **HTML Entity Escaping**: Apply HTML entity encoding to special characters using the `HTML_ESCAPE_TABLE` mapping. This prevents HTML injection and ensures proper display of literal angle brackets, ampersands, and quotes.
3. **Whitespace Preservation**: For code blocks and inline code elements, preserve all whitespace exactly as it appears in the source. For other elements, normalize whitespace according to HTML conventions while respecting intentional line breaks.

4. **URL Processing**: For link and image elements, validate and potentially transform URLs. This includes relative URL resolution, protocol validation, and optional URL rewriting for security or functionality purposes.
5. **Inline Content Integration**: Combine processed text content with the HTML generated from child inline elements, maintaining the correct order and nesting relationships established during inline parsing.

HTML Generator Architecture Decisions

The HTML generator design involves several critical architectural decisions that affect correctness, performance, and extensibility. Each decision represents a trade-off between competing concerns and establishes patterns that influence the entire generation system.

Decision: HTML Escaping Timing

- **Context**: HTML special characters (&, <, >, ", ') must be escaped to prevent malformed output and security vulnerabilities, but escaping must occur at the right time to avoid double-escaping or interference with legitimate HTML tags.
- **Options Considered**:
  1. Escape during initial text parsing before AST construction
  2. Escape during HTML generation at the leaf node level
  3. Escape at the very end after all HTML is generated
- **Decision**: Escape during HTML generation at the leaf node level when processing text content
- **Rationale**: This timing ensures that all text content is properly escaped while allowing legitimate HTML tags generated by the renderer to remain unescaped. Early escaping would interfere with inline parsing delimiter detection, while late escaping would risk missing content or double-escaping already-processed elements.
- **Consequences**: Enables safe handling of user content while maintaining clean separation between content and markup. Requires careful implementation to ensure all text paths go through escaping logic.

Option	Pros	Cons	Chosen?
Escape during parsing	Simple, happens once	Interferes with delimiter detection	No
Escape during generation	Clean separation, no double-escaping	Must be applied consistently	**Yes**
Escape after generation	Handles all content uniformly	Risk of escaping legitimate HTML tags	No

Decision: Self-Closing Tag Handling

- **Context**: HTML5 has specific rules for self-closing tags like `<hr>`, `<br>`, and `<img>` that should not have closing tags, while container tags like `<p>` and `<div>` must have matching closing tags.
- **Options Considered**:
  1. Always generate closing tags and let browsers handle invalid combinations
  2. Maintain a list of self-closing tag names and handle them specially
  3. Use XHTML-style self-closing syntax (`<br />`) for all self-closing elements
- **Decision**: Maintain a list of self-closing tag names and generate HTML5-compliant output
- **Rationale**: Produces the most correct and standards-compliant HTML output. Modern validation tools and accessibility checkers expect proper self-closing tag usage. XHTML syntax is unnecessarily verbose for HTML5 output.
- **Consequences**: Requires maintaining an accurate list of self-closing tags but produces the highest quality output. Enables proper validation and compatibility with HTML processing tools.

Option	Pros	Cons	Chosen?
Always generate closing tags	Simple implementation	Invalid HTML for self-closing elements	No
Special handling by tag name	Correct HTML5 output	Requires tag name knowledge	**Yes**
XHTML-style syntax	Valid in both HTML and XHTML	Unnecessarily verbose	No

Decision: Extensible Rendering Interface

- **Context**: Users may want to customize HTML output for specific node types, add custom attributes, or integrate with template systems while maintaining the core generation logic.
- **Options Considered**:
  1. Hard-coded rendering with no customization options
  2. Callback-based system where users can override specific node type rendering
  3. Template-based system where HTML generation uses external templates
- **Decision**: Callback-based system with a renderer plugin interface
- **Rationale**: Provides flexibility for customization while maintaining performance and simplicity. Templates would add complexity and external dependencies, while hard-coded rendering would limit extensibility for advanced users.
- **Consequences**: Enables custom output formats and integration with existing systems. Requires careful interface design to maintain backward compatibility and performance.

Option	Pros	Cons	Chosen?
Hard-coded rendering	Simple, fast, predictable	No customization possible	No
Callback-based plugins	Flexible, maintains performance	Interface complexity	**Yes**
Template-based system	Very flexible, familiar pattern	External dependencies, complexity	No

Decision: Pretty Printing Strategy

- **Context**: Generated HTML can be output as minified (single line) for efficiency or pretty-printed (indented, multi-line) for human readability, with implications for file size, debugging, and processing speed.
- **Options Considered**:
  1. Always generate minified output for smallest size
  2. Always generate pretty-printed output for readability
  3. Configurable option with default to pretty-printed
- **Decision**: Configurable option with default to pretty-printed output
- **Rationale**: Development and debugging benefit significantly from readable HTML output, while production systems can opt for minified output when needed. The size difference is typically negligible for markdown-generated content compared to other web assets.
- **Consequences**: Adds slight complexity to the generator but provides significant developer experience benefits. Pretty-printing logic must be carefully implemented to avoid introducing semantic whitespace changes.

Option	Pros	Cons	Chosen?
Always minified	Smallest output, fastest generation	Difficult to debug	No
Always pretty-printed	Easy debugging, readable output	Slightly larger files	No
Configurable with pretty default	Best of both worlds	Additional complexity	**Yes**

HTML Generation Component Responsibilities

The HTML generator consists of several specialized components, each with clearly defined responsibilities:

Component	Responsibilities	Dependencies	Output
`HtmlRenderer`	Orchestrates overall generation process	AST root node, configuration	Complete HTML document string
`NodeVisitor`	Implements depth-first tree traversal	AST node structure	Visits each node exactly once
`TagGenerator`	Creates opening and closing HTML tags	Node type mappings, attributes	Individual HTML tags with attributes
`ContentEscaper`	Applies HTML entity escaping to text	Escape mapping table	Safely escaped text content
`AttributeProcessor`	Converts node attributes to HTML attributes	Node attribute dictionaries	Formatted attribute strings
`PrettyPrinter`	Applies consistent formatting and indentation	Generated HTML string, formatting rules	Formatted HTML output
`RendererPlugin`	Provides extensibility for custom output	Plugin interface, node context	Custom HTML for specific nodes

HTML Generation Common Pitfalls

HTML generation involves several subtle issues that can produce incorrect output or security vulnerabilities. Understanding these pitfalls is crucial for implementing a robust and secure markdown renderer.

⚠️ Pitfall: Double-Escaping HTML Entities

The most common mistake in HTML generation is applying HTML entity escaping multiple times to the same content, resulting in output like `&lt;` instead of `<`. This occurs when text is escaped during parsing and then escaped again during HTML generation, or when already-escaped content from child nodes is re-escaped by parent nodes.

Why it's wrong: Double-escaped content displays incorrectly to users. For example, the HTML entity `<` should render as the `<` character, but `&lt;` renders as the literal text `<`.

How to avoid: Implement escaping exactly once during HTML generation when processing leaf text nodes. Ensure that content returned by child node processing is already properly escaped and should not be re-escaped. Use clear data flow contracts where escaped and unescaped content are never mixed without explicit conversion.

⚠️ Pitfall: Incorrect Self-Closing Tag Syntax

Many implementations generate invalid HTML by either adding closing tags to self-closing elements (`<hr></hr>`) or forgetting closing tags for container elements. This produces HTML that may render inconsistently across browsers and fails validation.

Why it's wrong: Invalid HTML can cause rendering differences between browsers, accessibility issues, and problems with HTML processing tools. Some browsers attempt to fix invalid markup automatically, leading to unpredictable results.

How to avoid: Maintain an accurate list of HTML5 self-closing elements (`hr`, `br`, `img`, `input`, `area`, `base`, `col`, `embed`, `link`, `meta`, `source`, `track`, `wbr`) and handle them specially in tag generation logic. Use a clear distinction in code between self-closing and container element generation.

⚠️ Pitfall: Whitespace Semantic Changes

Pretty-printing logic can inadvertently introduce or remove whitespace that has semantic meaning in HTML, particularly around inline elements where spaces between tags affect text layout and rendering.

Why it's wrong: Adding or removing spaces between inline elements changes how text flows and displays. For example, `<strong>bold</strong> <em>italic</em>` renders differently than `<strong>bold</strong><em>italic</em>` - the first has a space between "bold" and "italic" while the second does not.

How to avoid: Apply pretty-printing indentation only at block-level boundaries where whitespace is not semantically significant. Preserve exact spacing within and between inline elements. Test pretty-printed output against minified output to ensure semantic equivalence.

⚠️ Pitfall: Improper Attribute Value Escaping

HTML attributes have different escaping requirements than HTML content. Attribute values must escape quotes appropriately for the quote style used, and certain characters like newlines may need special handling or replacement.

Why it's wrong: Unescaped quotes in attribute values can break HTML parsing by prematurely terminating the attribute value. This can lead to HTML injection vulnerabilities or malformed output.

How to avoid: Use different escaping logic for attribute values than for HTML content. When using double quotes for attribute values, escape internal double quotes as `&quot;`. When using single quotes, escape internal single quotes as `&#39;`. Consider using double quotes consistently for all attributes to simplify escaping logic.

⚠️ Pitfall: Missing URL Validation and Escaping

Links and images require special handling for their URL attributes, including proper percent-encoding and validation to prevent malformed URLs or potential security issues with javascript: or data: URLs.

Why it's wrong: Invalid URLs can break page functionality or create security vulnerabilities. Unescaped special characters in URLs can cause parsing errors or unexpected behavior.

How to avoid: Implement URL-specific validation and encoding for `href` and `src` attributes. Consider implementing configurable URL scheme filtering to block potentially dangerous protocols. Apply proper percent-encoding to URL components while preserving valid URL structure.

⚠️ Pitfall: Inconsistent List Handling

Lists have complex rules around tight versus loose formatting, where tight lists render list items without paragraph tags while loose lists wrap list item content in paragraphs. Inconsistent handling produces incorrect HTML structure.

Why it's wrong: Incorrect list formatting affects both visual appearance and semantic meaning. Screen readers and other assistive technologies rely on proper list structure for navigation and content understanding.

How to avoid: Implement clear logic for determining list tightness based on blank lines between list items during parsing. Consistently apply paragraph wrapping rules based on the list's tight/loose status. Test with nested lists to ensure proper handling of mixed tight/loose scenarios.

⚠️ Pitfall: Plugin Interface State Pollution

When implementing extensible rendering through plugins, state sharing between the main renderer and plugins can cause unexpected interactions or side effects that affect subsequent rendering operations.

Why it's wrong: State pollution can cause rendering inconsistencies, memory leaks, or unexpected behavior when plugins modify shared state. This makes the system unreliable and difficult to debug.

How to avoid: Design plugin interfaces to be stateless where possible. Pass all necessary context as parameters rather than sharing mutable state. If state is required, use immutable data structures or clear ownership patterns where plugins cannot affect the main renderer's state.

Implementation Guidance

The HTML generation system requires careful coordination between several components while maintaining clean separation of concerns. The implementation focuses on correctness, security, and extensibility while providing clear debugging capabilities.

Technology Recommendations

Component	Simple Option	Advanced Option
HTML Escaping	Built-in string replacement with escape table	`html` module with `html.escape()` function
Tree Traversal	Recursive function with node type dispatch	Visitor pattern with abstract base classes
Template System	String formatting with f-strings	`jinja2` or similar template engine
Pretty Printing	Manual indentation tracking with string building	`xml.dom.minidom` for structured formatting
Plugin System	Simple callback dictionary	Full plugin architecture with registration
URL Validation	Basic regex pattern matching	`urllib.parse` with comprehensive validation

Recommended File Structure

The HTML generator integrates into the overall project structure as the final stage of the parsing pipeline:

```
project-root/
  src/
    markdown_renderer/
      core/
        ast_nodes.py              ← AST node definitions (from Data Model)
        parser.py                 ← Main parser orchestration
      parsers/
        block_parser.py           ← Block-level parsing logic
        inline_parser.py          ← Inline element parsing logic
        list_parser.py            ← List parsing logic
      generators/
        html_generator.py         ← Main HTML generation logic (THIS COMPONENT)
        html_escaper.py           ← HTML entity escaping utilities
        tag_generator.py          ← HTML tag creation logic
        pretty_printer.py         ← Output formatting utilities
        renderer_plugins.py       ← Extensible rendering interface
      utils/
        html_utils.py             ← HTML-specific utility functions
        url_validator.py          ← URL validation and encoding
  tests/
    test_html_generation.py       ← HTML generator test suite
    test_escaping.py             ← Entity escaping test cases
    test_plugins.py              ← Plugin system test cases
  examples/
    basic_usage.py               ← Simple HTML generation examples
    custom_renderer.py           ← Plugin usage examples
```

HTML Escaping Infrastructure (Complete Starter Code)

```python
"""HTML entity escaping utilities for safe content rendering."""

import re

from typing import Dict, Optional


# Complete mapping of characters that must be escaped in HTML content

HTML_ESCAPE_TABLE: Dict[str, str] = {

    '&': '&amp;',    # Must be first to avoid double-escaping

    '<': '&lt;',

    '>': '&gt;',

    '"': '&quot;',

    "'": '&#x27;',   # More compatible than &apos;

}


# Separate mapping for HTML attribute values

ATTRIBUTE_ESCAPE_TABLE: Dict[str, str] = {

    '&': '&amp;',

    '<': '&lt;',

    '>': '&gt;',

    '"': '&quot;',

    "'": '&#x27;',

    '\n': '&#10;',   # Newlines in attributes

    '\r': '&#13;',   # Carriage returns in attributes

    '\t': '&#9;',    # Tabs in attributes

}


class HtmlEscaper:

    """Handles HTML entity escaping for content and attributes."""



    def __init__(self):

        # Pre-compile regex patterns for performance

        self._content_pattern = re.compile('([' + ''.join(HTML_ESCAPE_TABLE.keys()) + '])')
```

```
        self._attr_pattern = re.compile('([' + ''.join(ATTRIBUTE_ESCAPE_TABLE.keys()) + '])')


    def escape_content(self, text: str) -> str:
        """Escape text content for safe inclusion in HTML."""
        if not text:
            return text

        def replace_char(match):
            return HTML_ESCAPE_TABLE[match.group(1)]

        return self._content_pattern.sub(replace_char, text)


    def escape_attribute(self, value: str) -> str:
        """Escape attribute value for safe inclusion in HTML attributes."""
        if not value:
            return value

        def replace_char(match):
            return ATTRIBUTE_ESCAPE_TABLE[match.group(1)]

        return self._attr_pattern.sub(replace_char, value)


    def is_safe_url_scheme(self, url: str) -> bool:
        """Check if URL uses a safe scheme for links and images."""
        safe_schemes = {'http', 'https', 'ftp', 'mailto', 'tel'}
        if ':' not in url:
            return True  # Relative URLs are safe

        scheme = url.split(':', 1)[0].lower()
        return scheme in safe_schemes
```

```
# Global escaper instance for convenience
html_escaper = HtmlEscaper()

def escape_html(text: str) -> str:
    """Convenience function for HTML content escaping."""
    return html_escaper.escape_content(text)

def escape_html_attribute(value: str) -> str:
    """Convenience function for HTML attribute escaping."""
    return html_escaper.escape_attribute(value)
```

Pretty Printing Utilities (Complete Starter Code)

```python
"""Pretty printing utilities for formatted HTML output."""

from typing import List, Optional

import re


class HtmlPrettyPrinter:

    """Formats HTML output with consistent indentation and line breaks."""



    def __init__(self, indent_size: int = 2, max_line_length: Optional[int] = None):

        self.indent_size = indent_size

        self.max_line_length = max_line_length



        # Elements that should have their content on the same line

        self.inline_elements = {

            'a', 'abbr', 'b', 'bdi', 'bdo', 'br', 'button', 'cite', 'code',

            'dfn', 'em', 'i', 'kbd', 'mark', 'q', 's', 'samp', 'small',

            'span', 'strong', 'sub', 'sup', 'time', 'u', 'var'

        }



        # Self-closing elements that don't have content

        self.void_elements = {

            'area', 'base', 'br', 'col', 'embed', 'hr', 'img', 'input',

            'link', 'meta', 'source', 'track', 'wbr'

        }



    def format_html(self, html: str, start_indent: int = 0) -> str:

        """Format HTML string with proper indentation and line breaks."""

        if not html.strip():

            return html



        lines = []
```

```
        current_indent = start_indent
        i = 0

        while i < len(html):
            if html[i] == '<':
                # Find the end of this tag
                tag_end = html.find('>', i)
                if tag_end == -1:
                    # Malformed HTML - just add the rest as-is
                    lines.append(' ' * (current_indent * self.indent_size) + html[i:])
                    break

                tag_content = html[i:tag_end + 1]

                # Determine if this is a closing tag, self-closing tag, or opening tag
                if tag_content.startswith('</'):
                    # Closing tag - decrease indent before adding
                    current_indent = max(0, current_indent - 1)
                    lines.append(' ' * (current_indent * self.indent_size) + tag_content)
                elif tag_content.endswith('/>') or self._is_void_element(tag_content):
                    # Self-closing or void element - no indent change
                    lines.append(' ' * (current_indent * self.indent_size) + tag_content)
                else:
                    # Opening tag - add then increase indent
                    lines.append(' ' * (current_indent * self.indent_size) + tag_content)
                    if not self._is_inline_element(tag_content):
                        current_indent += 1

                i = tag_end + 1
            else:
                # Text content - find the next tag or end of string
```

```
            next_tag = html.find('<', i)
            if next_tag == -1:
                text_content = html[i:].strip()
            else:
                text_content = html[i:next_tag].strip()

            if text_content:
                lines.append(' ' * (current_indent * self.indent_size) + text_content)

            i = next_tag if next_tag != -1 else len(html)

        return '\n'.join(lines)

    def _extract_tag_name(self, tag_content: str) -> str:
        """Extract tag name from tag content like '<div class="foo">' -> 'div'."""
        # Remove < and > brackets
        tag_inner = tag_content[1:-1].strip()
        if tag_inner.startswith('/'):
            tag_inner = tag_inner[1:]

        # Split on whitespace to get just the tag name
        return tag_inner.split()[0] if tag_inner else ''

    def _is_inline_element(self, tag_content: str) -> bool:
        """Check if tag represents an inline element."""
        tag_name = self._extract_tag_name(tag_content)
        return tag_name in self.inline_elements

    def _is_void_element(self, tag_content: str) -> bool:
        """Check if tag represents a void (self-closing) element."""
        tag_name = self._extract_tag_name(tag_content)
```

```
        return tag_name in self.void_elements


# Global pretty printer instance
html_pretty_printer = HtmlPrettyPrinter()


def format_html(html: str, indent_size: int = 2) -> str:
    """Convenience function for HTML pretty printing."""
    printer = HtmlPrettyPrinter(indent_size)
    return printer.format_html(html)
```

Core HTML Generation Logic (Skeleton with TODOs)

```python
"""Core HTML generation logic for converting AST to HTML."""

from typing import Dict, Any, List, Optional, Callable

from abc import ABC, abstractmethod


from ..core.ast_nodes import ASTNode, NodeType, BlockType, InlineType

from .html_escaper import html_escaper

from .pretty_printer import html_pretty_printer


class HtmlRenderer:

    """Main HTML generation class that converts AST to HTML output."""



    def __init__(self, pretty_print: bool = True, custom_renderers: Optional[Dict[str, Callable]] = None):

        self.pretty_print = pretty_print

        self.custom_renderers = custom_renderers or {}

        self.escaper = html_escaper

        self.pretty_printer = html_pretty_printer if pretty_print else None



    def render_to_html(self, ast_root: ASTNode) -> str:

        """Convert AST to complete HTML document string."""

        # TODO 1: Check if ast_root is None or empty and return appropriate default

        # TODO 2: Call render_node on the root to get the body content

        # TODO 3: If pretty_print is enabled, format the output using pretty_printer

        # TODO 4: Return the final HTML string

        # Hint: Consider whether to wrap in full HTML document or just return content

        pass



    def render_node(self, node: ASTNode) -> str:

        """Render a single AST node and all its children to HTML."""

        # TODO 1: Check if there's a custom renderer for this node type in self.custom_renderers

        # TODO 2: If custom renderer exists, call it with the node and return result

        # TODO 3: Dispatch to appropriate rendering method based on node.node_type
```

```
        # TODO 4: If node_type is BLOCK, call render_block_node(node)
        # TODO 5: If node_type is INLINE, call render_inline_node(node)
        # TODO 6: If node_type is TEXT, call render_text_node(node)
        # TODO 7: Handle unknown node types with graceful degradation
        # Hint: Use a dispatch dictionary to map node types to methods
        pass

    def render_block_node(self, node: ASTNode) -> str:
        """Render block-level elements like paragraphs, headings, lists."""
        # TODO 1: Extract block_type from node (it should be a BlockNode)
        # TODO 2: Dispatch based on block_type using if/elif or dictionary lookup
        # TODO 3: For PARAGRAPH: wrap children in <p> tags
        # TODO 4: For HEADING: determine level and wrap in <h1>-<h6> tags
        # TODO 5: For CODE_BLOCK: wrap in <pre><code> with language class if present
        # TODO 6: For BLOCKQUOTE: wrap in <blockquote> tags
        # TODO 7: For HORIZONTAL_RULE: return <hr> (self-closing)
        # TODO 8: For LIST: determine if ordered/unordered and wrap in <ol>/<ul>
        # TODO 9: For LIST_ITEM: wrap in <li> tags, handle tight/loose list formatting
        # TODO 10: Render all children and combine with appropriate wrapping
        # Hint: Use get_attribute() to access block-specific data like heading level
        pass

    def render_inline_node(self, node: ASTNode) -> str:
        """Render inline elements like emphasis, links, code spans."""
        # TODO 1: Extract inline_type from node (it should be an InlineNode)
        # TODO 2: Dispatch based on inline_type
        # TODO 3: For STRONG: wrap children in <strong> tags
        # TODO 4: For EMPHASIS: wrap children in <em> tags
        # TODO 5: For CODE: wrap text_content in <code> tags, preserve whitespace
        # TODO 6: For LINK: create <a> tag with href attribute, render children as link text
        # TODO 7: For IMAGE: create <img> tag with src and alt attributes (self-closing)
```

```
        # TODO 8: For LINE_BREAK: return <br> (self-closing)
        # TODO 9: Apply HTML escaping to text content but not to child HTML
        # TODO 10: Handle missing or invalid attributes gracefully
        # Hint: Use formatting_attributes dictionary for link URLs, image sources, etc.
        pass

    def render_text_node(self, node: ASTNode) -> str:
        """Render plain text nodes with proper HTML escaping."""
        # TODO 1: Extract text_content from the node
        # TODO 2: Handle None or empty text content by returning empty string
        # TODO 3: Apply HTML entity escaping using self.escaper.escape_content()
        # TODO 4: Return the escaped text (no HTML tags for text nodes)
        # Hint: This is where HTML injection prevention happens
        pass

    def render_children(self, node: ASTNode) -> str:
        """Render all child nodes and concatenate their HTML."""
        # TODO 1: Check if node has children attribute and it's not None
        # TODO 2: If no children, return empty string
        # TODO 3: Iterate through node.children list
        # TODO 4: Call render_node() recursively on each child
        # TODO 5: Concatenate all child HTML strings
        # TODO 6: Return the combined result
        # Hint: Maintain the order of children as they appear in the list
        pass

    def generate_tag(self, tag_name: str, attributes: Dict[str, str] = None,
                     self_closing: bool = False) -> tuple[str, str]:
        """Generate opening and closing HTML tags with attributes."""
        # TODO 1: Start building opening tag with '<' + tag_name
        # TODO 2: If attributes provided, iterate through them
```

```
        # TODO 3: For each attribute, escape the value using escaper.escape_attribute()
        # TODO 4: Add each attribute as ' key="escaped_value"' to opening tag
        # TODO 5: Close opening tag with '>' or '/>' if self_closing
        # TODO 6: If self_closing, return (opening_tag, '') for empty closing tag
        # TODO 7: If not self_closing, generate closing tag as '</' + tag_name + '>'
        # TODO 8: Return tuple of (opening_tag, closing_tag)
        # Hint: Check VOID_ELEMENTS list to determine if tag should be self-closing
        pass

# List of HTML void elements that should not have closing tags
VOID_ELEMENTS = {
    'area', 'base', 'br', 'col', 'embed', 'hr', 'img', 'input',
    'link', 'meta', 'source', 'track', 'wbr'
}

def render_ast_to_html(ast_root: ASTNode, pretty_print: bool = True) -> str:
    """Convenience function for rendering AST to HTML."""
    renderer = HtmlRenderer(pretty_print=pretty_print)
    return renderer.render_to_html(ast_root)
```

Plugin Interface Implementation (Complete Starter Code)

```python
"""Extensible rendering interface for custom HTML output."""

from typing import Protocol, Dict, Any, Optional, Callable

from abc import ABC, abstractmethod


from ..core.ast_nodes import ASTNode


class NodeRenderer(Protocol):

    """Protocol for custom node renderers."""



    def render(self, node: ASTNode, default_renderer: Callable[[ASTNode], str]) -> str:

        """Render a node with access to the default rendering logic."""

        ...


class RendererPlugin(ABC):

    """Base class for renderer plugins."""



    @abstractmethod

    def get_supported_types(self) -> List[str]:

        """Return list of node types this plugin can render."""

        pass



    @abstractmethod

    def render_node(self, node: ASTNode, context: Dict[str, Any]) -> str:

        """Render the given node type."""

        pass


class PluginManager:

    """Manages registration and execution of renderer plugins."""



    def __init__(self):

        self.plugins: Dict[str, RendererPlugin] = {}

        self.node_renderers: Dict[str, NodeRenderer] = {}
```

```
    def register_plugin(self, plugin: RendererPlugin):
        """Register a plugin for specific node types."""
        for node_type in plugin.get_supported_types():
            self.plugins[node_type] = plugin

    def register_node_renderer(self, node_type: str, renderer: NodeRenderer):
        """Register a simple node renderer function."""
        self.node_renderers[node_type] = renderer

    def has_renderer(self, node_type: str) -> bool:
        """Check if a custom renderer exists for the node type."""
        return node_type in self.plugins or node_type in self.node_renderers

    def render_node(self, node_type: str, node: ASTNode,
                    context: Dict[str, Any], default_renderer: Callable) -> str:
        """Render node using custom renderer if available."""
        if node_type in self.node_renderers:
            return self.node_renderers[node_type].render(node, default_renderer)
        elif node_type in self.plugins:
            return self.plugins[node_type].render_node(node, context)
        else:
            return default_renderer(node)

# Example custom renderer for code blocks with syntax highlighting
class SyntaxHighlightRenderer(RendererPlugin):
    """Example plugin that adds syntax highlighting to code blocks."""

    def get_supported_types(self) -> List[str]:
        return ['CODE_BLOCK']
```

```
    def render_node(self, node: ASTNode, context: Dict[str, Any]) -> str:
        # This is a simplified example - real syntax highlighting
        # would integrate with libraries like Pygments
        language = node.get_attribute('language', '')
        content = self.escape_html(node.get_attribute('content', ''))

        if language:
            return f'<pre><code class="language-{language} highlighted">{content}</code></pre>'
        else:
            return f'<pre><code class="highlighted">{content}</code></pre>'

    def escape_html(self, text: str) -> str:
        """Simple HTML escaping for code content."""
        return (text.replace('&', '&amp;')
                    .replace('<', '&lt;')
                    .replace('>', '&gt;')
                    .replace('"', '&quot;')
                    .replace("'", '&#x27;'))
```

Milestone Checkpoint

After implementing the HTML generator, verify your implementation with these checkpoints:

1. **Basic HTML Generation Test**: Create a simple AST with a paragraph containing text and verify it generates `<p>escaped text content</p>`.
2. **Escaping Verification**: Test that input containing `<script>` and other HTML special characters are properly escaped in the output.
3. **Self-Closing Tags Test**: Verify that `<hr>` elements are generated without closing tags and `<img>` elements have proper `src` and `alt` attributes.
4. **Pretty Printing Test**: Generate HTML with pretty printing enabled and verify consistent indentation and line breaks.
5. **Plugin System Test**: Register a custom renderer for headings and verify it's called instead of the default renderer.

Expected output for a basic test case:

```HTML
<h1>Test Heading</h1>

<p>This is a paragraph with <strong>bold text</strong> and a <a href="http://example.com">link</a>.</p>

<hr>

<pre><code class="language-python">print("Hello, world!")</code></pre>
```

Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
Output contains `&lt;` instead of `<`	Double-escaping of content	Check if content is escaped multiple times	Apply escaping only once during HTML generation
Invalid HTML with `<hr></hr>` tags	Self-closing elements getting closing tags	Verify void element list and tag generation logic	Add proper self-closing tag detection
Missing spaces between inline elements	Pretty printer removing semantic whitespace	Compare pretty-printed vs minified output	Preserve whitespace around inline boundaries
Links or images not working	Unescaped or invalid URLs in attributes	Inspect generated `href` and `src` attributes	Apply proper URL validation and attribute escaping
Plugin not being called	Registration or dispatch logic issues	Add debug logging to plugin manager	Verify plugin registration and node type matching
Malformed nested HTML	Incorrect tree traversal or tag matching	Print AST structure and trace rendering order	Fix recursive child rendering and tag pairing

Interactions and Data Flow

> **Milestone(s):** Milestone 1: Block Elements, Milestone 2: Inline Elements, Milestone 3: Lists, Milestone 4: HTML Generation

The Orchestra Conductor Mental Model

Think of the markdown parsing pipeline like a symphony orchestra performing a complex piece. Each component—the block parser, inline parser, and HTML generator—is like a different section of the orchestra (strings, woodwinds, brass). The `MarkdownParser` acts as the conductor, coordinating when each section plays and ensuring the data flows smoothly from one to the next. Just as the conductor ensures the violins finish their phrase before the trumpets begin, the parser ensures each phase completes and passes clean, well-structured data to the next phase. The AST serves as the musical score—a shared representation that all components can read and contribute to.

The beauty of this orchestrated approach is that each component can focus on its specialized task without worrying about the complexities of other phases. The block parser doesn't need to understand emphasis rules, just as the first violin doesn't need to know the tuba part. But through careful coordination and clear interfaces, they work together to create something beautiful and coherent.

Complete Parsing Pipeline

The markdown parsing pipeline transforms raw text through four distinct phases, each building upon the work of the previous phase. This sequential approach mirrors how humans naturally process documents—first identifying the major structural elements, then focusing on the formatting details within those structures.

Pipeline Phase Overview

The complete transformation follows this sequence:

1. **Preprocessing Phase**: Raw markdown text undergoes normalization to ensure consistent line endings, tab expansion, and basic structural analysis
2. **Block Parsing Phase**: Normalized text is analyzed line-by-line to identify and construct block-level elements like headings, paragraphs, and code blocks
3. **Inline Parsing Phase**: Text content within block elements is processed to identify and construct inline formatting like emphasis, links, and inline code
4. **HTML Generation Phase**: The completed AST is traversed to generate clean, properly escaped HTML output

Phase	Input Format	Output Format	Primary Responsibility	State Maintained
Preprocessing	Raw string	List of `LineInfo` objects	Text normalization, line analysis	None (stateless)
Block Parsing	`LineInfo` objects	AST with block structure	Block element identification	`BlockParserState`, current block context
Inline Parsing	Block nodes with raw text	AST with inline elements	Inline formatting identification	`DelimiterStack`, escape processing state
HTML Generation	Complete AST	HTML string	Tree traversal and HTML generation	Rendering context, indentation state

Detailed Pipeline Flow

Preprocessing Phase: The `Preprocessor` receives the raw markdown text and performs essential normalization operations. It calls `normalize_line_endings()` to convert all line endings to Unix format, ensuring consistent behavior across platforms. The `split_into_lines()` function then breaks the text into individual lines while preserving line number information for error reporting. Each line is analyzed using `is_blank_line()` and `get_indentation_level()` to create rich `LineInfo` objects that carry both content and metadata.

> The preprocessing phase is crucial because it establishes the foundation for all subsequent parsing. By normalizing text encoding and capturing structural metadata early, we prevent countless edge cases that would otherwise plague the parsing phases.

Block Parsing Phase: The block parser processes the `LineInfo` sequence through a state machine that identifies block-level structures. It maintains a `BlockParserState` that tracks whether it's currently inside a paragraph, code block, blockquote, or looking for a new block to start. The parser calls `peek_next_line()` to look ahead when necessary, particularly for Setext headings that require examining the following line. Each line triggers either `can_current_block_continue()` to extend the current block or one of the `try_start_*_block()` methods to begin a new block type.

The block parser builds the AST incrementally, creating `BlockNode` instances and establishing parent-child relationships through the `add_child()` method. When a block is complete, `finalize_current_block()` processes any remaining content and adds the block to the document tree.

Inline Parsing Phase: Once the block structure is established, the inline parser processes the text content within each block element. It operates on individual text spans, typically paragraph content or heading text, using `parse_inline_elements()` as the main entry point. The parser maintains a `DelimiterStack` to track potential formatting markers as it scans through the text character by character.

When the parser encounters potential delimiters like asterisks or underscores, it uses `detect_flanking_properties()` to determine whether the delimiter can open or close emphasis. The `find_matching_opener()` function searches back through the delimiter stack to find valid emphasis pairs. Links and images are handled through the `LinkImageParser`, which uses specialized regex patterns to extract URLs and alt text.

HTML Generation Phase: The final phase traverses the complete AST using depth-first traversal to generate HTML output. The `HtmlRenderer` calls `render_node()` for each AST node, which delegates to specialized methods like `render_block_node()` and `render_inline_node()` based on the node type. The `HtmlEscaper` ensures all text content is properly escaped using `escape_html()`, while `generate_tag()` creates well-formed HTML elements with appropriate attributes.

Pipeline Error Handling and Recovery

Each phase implements graceful degradation when encountering malformed input. The preprocessing phase handles unusual line ending combinations and encoding issues by falling back to best-effort normalization. The block parser implements error recovery by treating unrecognized input as plain paragraph content. The inline parser handles mismatched delimiters by treating them as literal text. The HTML generator ensures output is always valid HTML, even when the AST contains unexpected structures.

Error Scenario	Detection Phase	Recovery Strategy	Output Impact
Invalid UTF-8 sequences	Preprocessing	Replace with Unicode replacement character	Minimal - content preserved
Unclosed code fences	Block parsing	Treat as paragraph content	Moderate - formatting lost but readable
Mismatched emphasis delimiters	Inline parsing	Render delimiters as literal text	Minimal - content fully preserved
Malformed link syntax	Inline parsing	Render as plain text	Minimal - text content preserved
Missing AST node types	HTML generation	Render as plain text with warning	Low - graceful fallback

Data Flow Contracts Between Phases

Each pipeline phase establishes clear contracts for the data it receives and produces. These contracts ensure that phases can evolve independently while maintaining system integrity.

Preprocessing → Block Parsing Contract: The preprocessor guarantees that line endings are normalized, tab characters are expanded to spaces using `expand_tabs_to_spaces()`, and each `LineInfo` object contains accurate `line_number`, `content`, `is_blank`, and `indent_level` fields. The block parser can rely on this normalized format and doesn't need to handle platform-specific text variations.

Block Parsing → Inline Parsing Contract: The block parser guarantees that the AST contains a valid tree structure with proper parent-child relationships established through `add_child()` calls. Each `BlockNode` contains raw text content in its `inline_content` field that represents the content to be processed for inline elements. List items, headings, and paragraph blocks all provide their text content in a consistent format that the inline parser can process uniformly.

Inline Parsing → HTML Generation Contract: The inline parser guarantees that all text content within block elements has been fully processed for inline formatting. The AST contains a complete representation of the document structure, with `InlineNode` instances properly nested within their containing blocks. The HTML generator can traverse this tree without needing to understand markdown syntax, focusing purely on HTML generation concerns.

Component Interface Contracts

The parsing pipeline depends on well-defined interfaces between components to maintain modularity and enable independent testing. These contracts specify not just method signatures but also preconditions, postconditions, and behavioral guarantees.

MarkdownParser Primary Interface

The `MarkdownParser` serves as the main orchestrator, providing the primary entry points that applications use to transform markdown text into HTML.

Method Name	Parameters	Returns	Description	Preconditions	Postconditions
`parse_to_html`	`markdown_text: str`	`str`	Complete markdown to HTML transformation	Text must be valid Unicode string	Returns valid HTML5 document fragment
`parse_file`	`file_path: str`	`str`	File-based parsing with error handling	File must exist and be readable	Returns HTML or raises IOException
`debug_print_ast`	`node: ASTNode, indent: int`	`None`	Display AST structure for debugging	Node must be valid AST structure	Prints tree to stdout

The `parse_to_html()` method orchestrates the entire pipeline. It first calls the preprocessor's `process_input()` method to normalize the input text. The resulting `LineInfo` objects are passed to the block parser's `parse_blocks()` method, which returns an AST with block structure. Each block node's text content is then processed by the inline parser's `parse_inline_elements()` method. Finally, the complete AST is passed to the HTML generator's `render_to_html()` method.

Decision: Sequential Pipeline Architecture

- **Context**: We need to coordinate multiple parsing phases that have dependencies on each other's output
- **Options Considered**:
  1. Monolithic parser that handles all concerns simultaneously
  2. Sequential pipeline with clear phase boundaries
  3. Concurrent pipeline with synchronization points
- **Decision**: Sequential pipeline with clear phase boundaries
- **Rationale**: Block parsing must complete before inline parsing can begin, since inline parsing depends on knowing the block context. This natural dependency makes sequential processing the most straightforward approach. The overhead of multiple passes is minimal compared to the complexity savings.
- **Consequences**: Enables clear separation of concerns and independent testing of each phase, but requires multiple passes through the document structure

Preprocessor Interface Contract

The `Preprocessor` provides text normalization services that prepare raw markdown for block parsing. This component handles platform differences and encoding issues that would otherwise complicate the parsing phases.

Method Name	Parameters	Returns	Description	Guarantees
`process_input`	`text: str`	`List[LineInfo]`	Convert raw text to analyzed lines	Line endings normalized, indentation calculated
`normalize_line_endings`	`text: str`	`str`	Convert all line endings to Unix format	Only \n line endings in output
`split_into_lines`	`text: str`	`List[str]`	Split text preserving empty lines	Line count preserved, no content lost
`expand_tabs_to_spaces`	`text: str, tab_width: int`	`str`	Convert tabs to spaces	Tab stops respect CommonMark spec

The `process_input()` method serves as the main entry point, combining all normalization operations into a single call. It ensures that downstream components receive consistent, well-formed input regardless of the original text encoding or platform conventions.

Block Parser Interface Contract

The block parser transforms normalized text into an AST representing the document's block structure. This component implements the most complex parsing logic, handling state transitions and lookahead requirements.

Method Name	Parameters	Returns	Description	State Changes
`parse_blocks`	`markdown_text: str`	`ASTNode`	Main block parsing entry point	Creates new parser state
`process_line_sequence`	`lines: List[LineInfo]`	`None`	Process lines through state machine	Updates `BlockParserState`
`peek_next_line`	None	`Optional[LineInfo]`	Look ahead without consuming	No state changes
`can_current_block_continue`	`line: LineInfo`	`bool`	Check if line extends current block	No state changes
`try_start_heading_block`	`line: LineInfo`	`bool`	Attempt to create heading block	May create new block
`try_start_code_block`	`line: LineInfo`	`bool`	Attempt to create code block	May create new block
`try_start_blockquote`	`line: LineInfo`	`bool`	Attempt to create blockquote	May create new block
`try_start_horizontal_rule`	`line: LineInfo`	`bool`	Attempt to create horizontal rule	May create new block
`start_paragraph_block`	`line: LineInfo`	`None`	Start paragraph with given line	Creates paragraph block
`continue_current_block`	`line: LineInfo`	`None`	Add line to active block	Updates current block content
`finalize_current_block`	None	`None`	Complete current block	Adds block to AST, resets state

The block parser maintains internal state through the `BlockParserState` enum, which tracks whether the parser is looking for a new block, continuing a paragraph, or inside a specialized block like a code block or blockquote. The

`try_start_*_block()` methods implement the block recognition logic, checking line patterns against the various block type signatures defined in the CommonMark specification.

> The block parser's lookahead capability through `peek_next_line()` is essential for handling Setext headings, where a line's meaning depends on the following line. This lookahead is implemented without consuming the line from the input stream, allowing the parser to make decisions while preserving the sequential processing model.

Inline Parser Interface Contract

The inline parser processes text content within block elements to identify and construct inline formatting elements. This component handles the complex delimiter matching required for nested emphasis and link parsing.

Method Name	Parameters	Returns	Description	State Management
`parse_inline_elements`	`text: str`	`List[InlineNode]`	Main inline parsing entry point	Creates fresh delimiter stack
`process_escapes`	`text: str`	`Tuple[str, List[int]]`	Process escape sequences	Returns processed text and positions
`detect_flanking_properties`	`text: str, pos: int, delimiter_len: int`	`Tuple[bool, bool]`	Detect opener/closer capability	No state changes
`find_matching_opener`	`closer: Delimiter`	`Optional[Tuple[int, Delimiter]]`	Find matching emphasis opener	Searches delimiter stack
`can_open_emphasis`	`char: str, left_flanking: bool, right_flanking: bool, before_char: str, after_char: str`	`bool`	Check if delimiter can open	Uses flanking detection rules
`can_close_emphasis`	`char: str, left_flanking: bool, right_flanking: bool, before_char: str, after_char: str`	`bool`	Check if delimiter can close	Uses flanking detection rules

The inline parser uses a `DelimiterStack` to track potential formatting markers as it processes text character by character. When it encounters asterisks, underscores, or other delimiter characters, it pushes them onto the stack with metadata about their flanking properties. When it finds potential closing delimiters, it searches backward through the stack to find matching openers.

The `LinkImageParser` component handles the complex regular expression matching required for link and image syntax. It maintains compiled patterns for URL detection and title extraction, ensuring efficient processing of link-heavy documents.

HTML Generator Interface Contract

The HTML generator traverses the completed AST to produce clean, well-formatted HTML output. This component handles HTML escaping, tag generation, and pretty printing concerns.

Method Name	Parameters	Returns	Description	Output Guarantees
`render_to_html`	`ast_root: ASTNode`	`str`	Main HTML generation entry point	Valid HTML5 fragment
`render_node`	`node: ASTNode`	`str`	Render single AST node	Properly escaped HTML
`render_block_node`	`node: BlockNode`	`str`	Render block-level elements	Block-level HTML tags
`render_inline_node`	`node: InlineNode`	`str`	Render inline elements	Inline HTML tags
`render_text_node`	`node: ASTNode`	`str`	Render plain text with escaping	HTML entity escaped text
`render_children`	`node: ASTNode`	`str`	Render all child nodes	Concatenated child HTML
`generate_tag`	`tag_name: str, attributes: dict, self_closing: bool`	`Tuple[str, str]`	Create HTML tag pair	Well-formed opening/closing tags
`escape_html`	`text: str`	`str`	Escape special characters	HTML entity escaped output
`escape_html_attribute`	`value: str`	`str`	Escape attribute values	Attribute-safe escaped text
`format_html`	`html: str, indent_size: int`	`str`	Pretty print HTML output	Indented, formatted HTML

The HTML generator implements the visitor pattern for tree traversal, calling the appropriate `render_*_node()` method based on each node's type. The `HtmlEscaper` component handles the critical security concern of properly escaping user content to prevent HTML injection attacks.

Decision: Visitor Pattern for HTML Generation

- **Context**: Need to traverse an AST with multiple node types and generate appropriate HTML for each type
- **Options Considered**:
  1. Switch statement based on node type in single method
  2. Visitor pattern with specialized methods for each node type
  3. Node-specific render methods on each AST node class
- **Decision**: Visitor pattern with specialized render methods
- **Rationale**: Keeps rendering logic centralized in the HTML generator while allowing easy extension for new node types. Avoids polluting AST nodes with rendering concerns.
- **Consequences**: Clear separation of concerns and easy testing, but requires coordinated changes when adding new node types

Plugin and Extension Interface

The HTML generator supports customization through a plugin interface that allows users to override rendering behavior for specific node types.

Method Name	Parameters	Returns	Description	Extension Point
`register_plugin`	`plugin: RendererPlugin`	`None`	Register custom renderer plugin	Adds to plugin registry
`register_node_renderer`	`node_type: str, renderer: NodeRenderer`	`None`	Register simple node renderer	Overrides default rendering
`has_renderer`	`node_type: str`	`bool`	Check for custom renderer	Query capability

The `PluginManager` maintains a registry of custom renderers that can override the default HTML generation for specific node types. This allows applications to customize output format (e.g., generating LaTeX instead of HTML) or add special handling for domain-specific markdown extensions.

Error Propagation and Recovery Contracts

Each component defines clear error handling contracts that specify when errors are recoverable versus when they should propagate to the caller.

Component	Recoverable Errors	Non-Recoverable Errors	Recovery Strategy
Preprocessor	Unusual line endings, mixed encodings	Invalid UTF-8, null input	Best-effort normalization
Block Parser	Unclosed blocks, invalid syntax	Null input, corrupted AST	Treat as paragraph content
Inline Parser	Mismatched delimiters, malformed links	Null text, invalid AST nodes	Render as literal text
HTML Generator	Unknown node types, missing attributes	Null AST, circular references	Default rendering with warnings

This error handling strategy ensures that the parser exhibits graceful degradation—it always produces some meaningful output, even when the input contains syntax errors or unexpected constructs.

Implementation Guidance

Technology Recommendations

Component	Simple Implementation	Advanced Implementation
Main Parser	Single class with delegation methods	Factory pattern with dependency injection
AST Representation	Simple dataclass/struct with lists	Immutable nodes with persistent data structures
Error Handling	Exception propagation with try/catch	Result/Either types with error accumulation
Plugin System	Dictionary of callable functions	Full plugin architecture with lifecycle management
Testing Infrastructure	unittest with manual test data	Property-based testing with generated inputs

Recommended File Structure

```
markdown_renderer/
  __init__.py                  ← Public API exports
  parser.py                    ← Main MarkdownParser class
  preprocessor.py              ← Text normalization utilities
  ast_nodes.py                 ← AST node type definitions
  block_parser.py              ← Block parsing implementation
  inline_parser.py             ← Inline parsing implementation
  html_generator.py            ← HTML generation and rendering
  plugins/
    __init__.py                ← Plugin interface definitions
    base_plugin.py             ← Abstract base classes
  utils/
    __init__.py
    line_info.py               ← LineInfo and text utilities
    html_escaping.py           ← HTML escaping utilities
    constants.py               ← Regex patterns and constants
  tests/
    test_integration.py        ← End-to-end pipeline tests
    test_parser_orchestration.py ← Component interaction tests
    test_error_handling.py     ← Error recovery tests
```

Main Parser Infrastructure Code

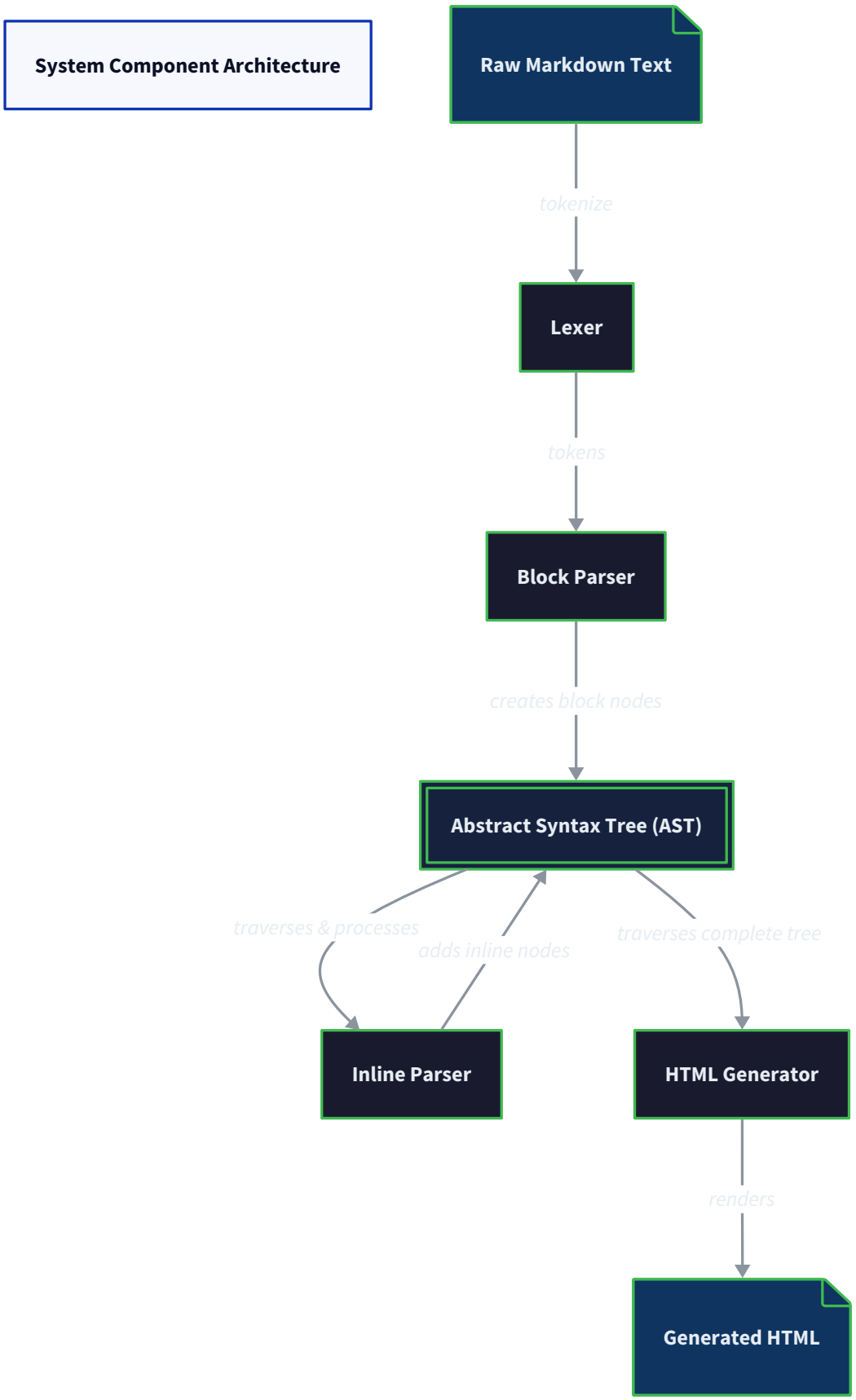
```python
# parser.py - Complete main orchestrator implementation

from typing import Optional, List, Union, Protocol

import re

from dataclasses import dataclass


from .preprocessor import Preprocessor

from .block_parser import BlockParser

from .inline_parser import InlineParser

from .html_generator import HtmlRenderer

from .ast_nodes import ASTNode, NodeType

from .utils.line_info import LineInfo


class MarkdownParser:

    """Main parser that orchestrates the complete markdown to HTML pipeline."""



    def __init__(self,

                 pretty_print: bool = True,

                 enable_plugins: bool = False):

        """Initialize the markdown parser with component dependencies.



        Args:

            pretty_print: Whether to format HTML output with indentation

            enable_plugins: Whether to load and enable plugin system

        """

        self.preprocessor = Preprocessor()

        self.block_parser = BlockParser()

        self.inline_parser = InlineParser()

        self.html_generator = HtmlRenderer(

            pretty_print=pretty_print,

            custom_renderers={} if not enable_plugins else None

        )
```

```
    def parse_to_html(self, markdown_text: str) -> str:
        """Transform markdown text to HTML through complete pipeline.

        Args:
            markdown_text: Raw markdown content as Unicode string

        Returns:
            Valid HTML5 document fragment

        Raises:
            ValueError: If input is None or contains invalid UTF-8
        """
        if markdown_text is None:
            raise ValueError("Markdown text cannot be None")

        try:
            # Phase 1: Preprocess and normalize input
            line_info_list = self.preprocessor.process_input(markdown_text)

            # Phase 2: Parse block structure
            ast_root = self.block_parser.parse_blocks(line_info_list)

            # Phase 3: Parse inline elements within blocks
            self._process_inline_content(ast_root)

            # Phase 4: Generate HTML output
            return self.html_generator.render_to_html(ast_root)

        except Exception as e:
            # Log error and attempt graceful degradation
```

```
        return f"<p>Error processing markdown: {self._escape_error_message(str(e))}</p>"


def parse_file(self, file_path: str) -> str:
    """Parse markdown file and return HTML.

    Args:
        file_path: Path to markdown file

    Returns:
        HTML string or error message

    Raises:
        IOError: If file cannot be read
    """
    try:
        with open(file_path, 'r', encoding='utf-8') as f:
            content = f.read()
        return self.parse_to_html(content)
    except IOError as e:
        raise IOError(f"Cannot read file {file_path}: {e}")


def debug_print_ast(self, node: ASTNode, indent: int = 0) -> None:
    """Print AST structure for debugging purposes.

    Args:
        node: Root node to print
        indent: Current indentation level
    """
    indent_str = "  " * indent
    node_info = f"{node.node_type.value}"
```

```
        if hasattr(node, 'block_type'):
            node_info += f" ({node.block_type.value})"
        elif hasattr(node, 'inline_type'):
            node_info += f" ({node.inline_type.value})"

        print(f"{indent_str}{node_info}")

        for child in node.children:
            self.debug_print_ast(child, indent + 1)

    def _process_inline_content(self, ast_root: ASTNode) -> None:
        """Process inline content for all applicable block nodes."""
        # TODO 1: Traverse AST using depth_first_walk with inline processing visitor
        # TODO 2: For each block node, check if it contains inline content
        # TODO 3: Call inline_parser.parse_inline_elements on block's text content
        # TODO 4: Replace block's text content with parsed inline node children
        # TODO 5: Handle error cases where inline parsing fails
        pass

    def _escape_error_message(self, message: str) -> str:
        """Escape error message for safe HTML inclusion."""
        return (message.replace('&', '&amp;')
                       .replace('<', '&lt;')
                       .replace('>', '&gt;')
                       .replace('"', '&quot;'))

# Public API convenience functions
def parse_to_html(markdown_text: str, pretty_print: bool = True) -> str:
    """Convenience function for simple markdown to HTML conversion."""
    parser = MarkdownParser(pretty_print=pretty_print)
    return parser.parse_to_html(markdown_text)
```

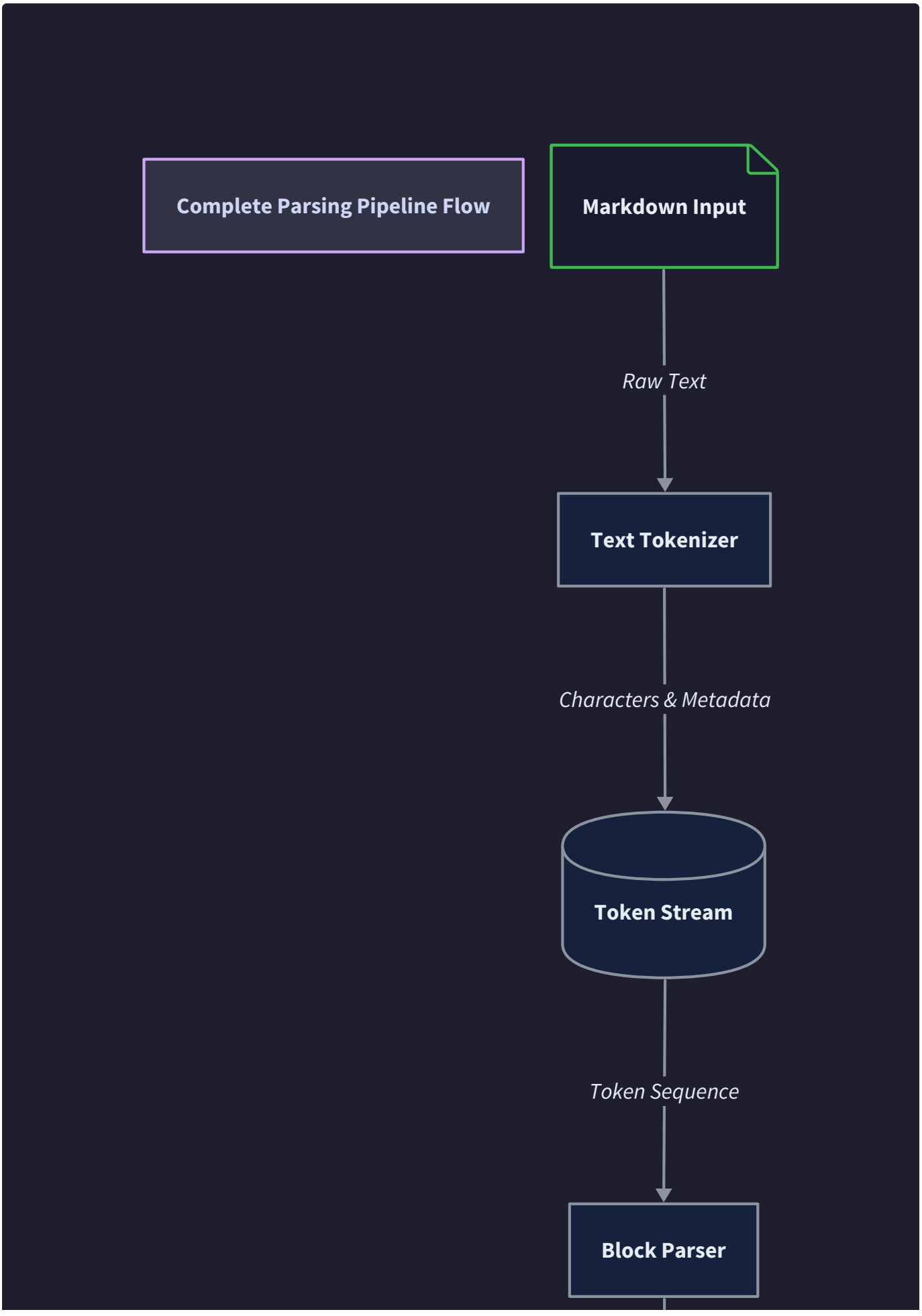
```
def parse_file(file_path: str, pretty_print: bool = True) -> str:
    """Convenience function for file-based markdown conversion."""
    parser = MarkdownParser(pretty_print=pretty_print)
    return parser.parse_file(file_path)
```

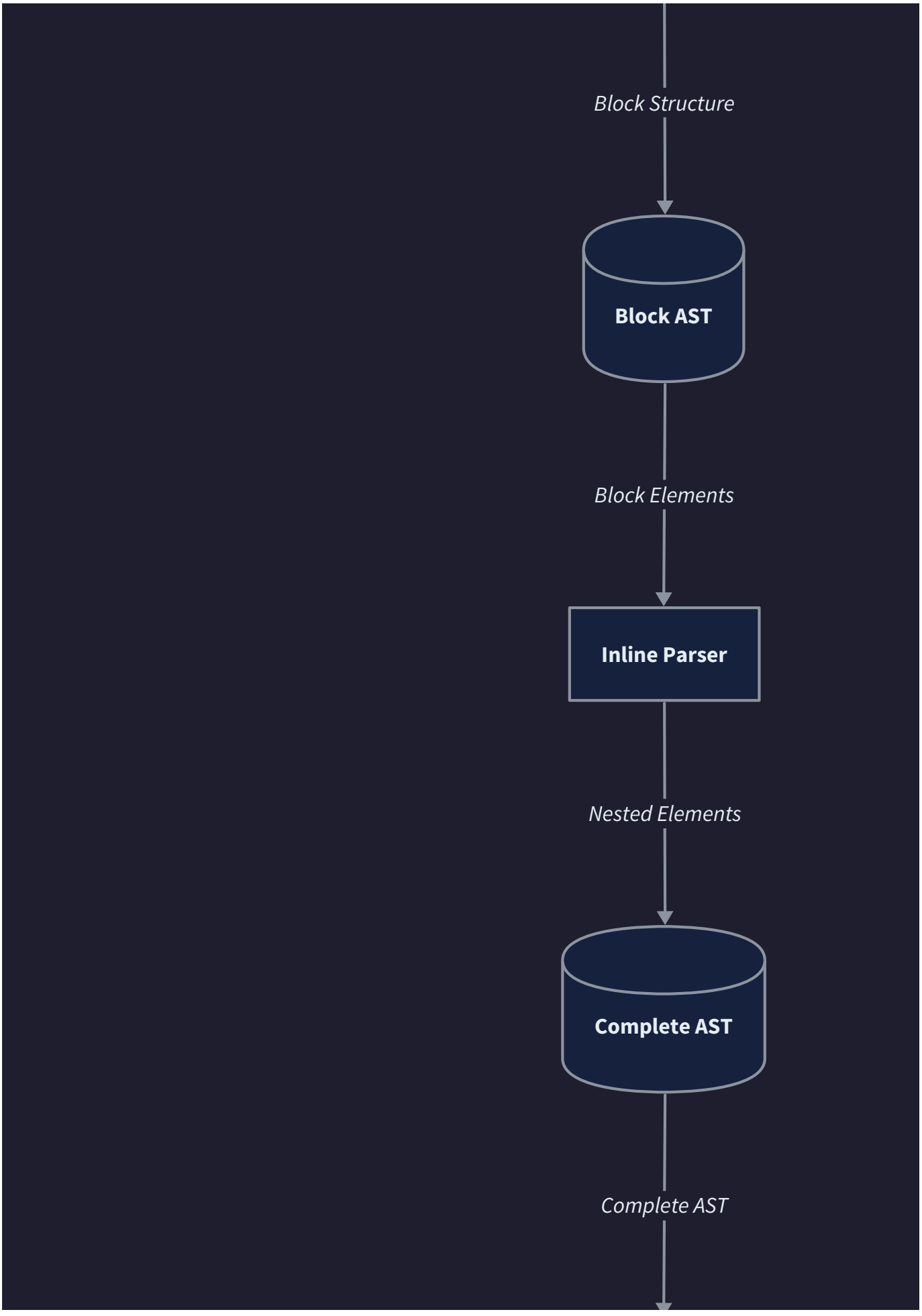

System Component Architecture
Raw Markdown Text
tokenize
Lexer
tokens
Block Parser
creates block nodes
Abstract Syntax Tree (AST)
traverses & processes
adds inline nodes
traverses complete tree
Inline Parser
HTML Generator
renders
Generated HTML

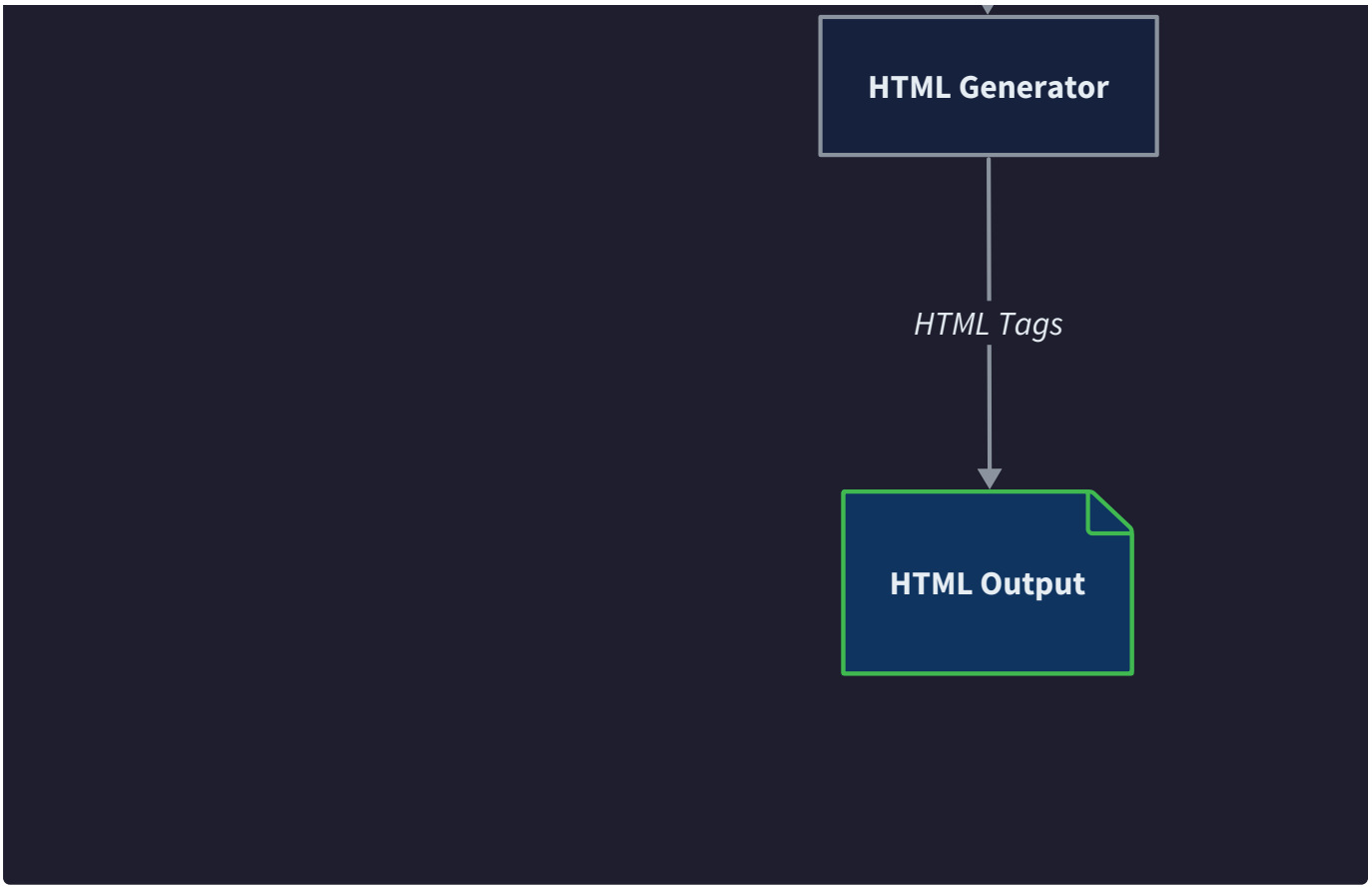
Pipeline Orchestration Skeleton

```python
# pipeline.py - Core pipeline coordination logic (implement this)

from typing import List, Iterator

from .ast_nodes import ASTNode, BlockNode, NodeType

from .utils.line_info import LineInfo


class PipelineOrchestrator:

    """Coordinates the flow of data between parsing phases."""



    def execute_complete_pipeline(self,

                                  raw_text: str,

                                  preprocessor,

                                  block_parser,

                                  inline_parser,

                                  html_generator) -> str:

        """Execute complete parsing pipeline with error recovery.



        This is the core implementation you need to complete.

        """

        # TODO 1: Validate input text is not None/empty

        # TODO 2: Call preprocessor.process_input() to get LineInfo list

        # TODO 3: Call block_parser.parse_blocks() with LineInfo list

        # TODO 4: Walk through AST calling process_block_inline_content()

        # TODO 5: Call html_generator.render_to_html() on completed AST

        # TODO 6: Handle any exceptions with graceful degradation

        # TODO 7: Return final HTML string

        pass



    def process_block_inline_content(self,

                                     block_node: BlockNode,
```

```
                                inline_parser) -> None:
        """Process inline content within a single block node."""
        # TODO 1: Check if block_node has inline_content field
        # TODO 2: Skip processing if content is empty or None
        # TODO 3: Call inline_parser.parse_inline_elements() on content
        # TODO 4: Add returned InlineNode objects as children to block_node
        # TODO 5: Clear the inline_content field since it's now parsed
        # TODO 6: Recursively process any child blocks
        pass


    def validate_pipeline_state(self,
                                ast_root: ASTNode) -> List[str]:
        """Validate AST structure before HTML generation."""
        # TODO 1: Check that ast_root is not None and has valid node_type
        # TODO 2: Verify all BlockNode objects have either children or content
        # TODO 3: Verify all parent-child relationships are bidirectional
        # TODO 4: Check for circular references in AST structure
        # TODO 5: Return list of validation errors (empty list if valid)
        pass
```


Complete Parsing Pipeline Flow
Markdown Input
Raw Text
Text Tokenizer
Characters & Metadata
Token Stream
Token Sequence
Block Parser


Block Structure
Block AST
Block Elements
Inline Parser
Nested Elements
Complete AST
Complete AST



Error Recovery Infrastructure

PYTHON

```python
# error_recovery.py - Complete error handling utilities

import logging

from typing import Optional, Any, List, Union

from enum import Enum


class ErrorSeverity(Enum):

    WARNING = "warning"

    ERROR = "error"

    CRITICAL = "critical"


class ParseError:

    """Structured representation of parsing errors."""


    def __init__(self,

                 message: str,

                 line_number: Optional[int] = None,

                 column: Optional[int] = None,
```

```
                 severity: ErrorSeverity = ErrorSeverity.ERROR):
        self.message = message
        self.line_number = line_number
        self.column = column
        self.severity = severity


    def __str__(self) -> str:
        location = ""
        if self.line_number is not None:
            location = f" at line {self.line_number}"
            if self.column is not None:
                location += f", column {self.column}"
        return f"{self.severity.value.upper()}: {self.message}{location}"

class ErrorCollector:
    """Collects and manages parsing errors during pipeline execution."""


    def __init__(self):
        self.errors: List[ParseError] = []
        self.logger = logging.getLogger(__name__)


    def add_error(self,
                  message: str,
                  line_number: Optional[int] = None,
                  column: Optional[int] = None,
                  severity: ErrorSeverity = ErrorSeverity.ERROR) -> None:
        """Add a parsing error to the collection."""
        error = ParseError(message, line_number, column, severity)
        self.errors.append(error)
        self.logger.log(self._severity_to_log_level(severity), str(error))
```

```
    def has_critical_errors(self) -> bool:
        """Check if any critical errors were collected."""
        return any(error.severity == ErrorSeverity.CRITICAL for error in self.errors)


    def get_error_summary(self) -> str:
        """Generate human-readable error summary."""
        if not self.errors:
            return "No errors"

        summary = f"Found {len(self.errors)} parsing issues:\n"
        for error in self.errors:
            summary += f"  - {error}\n"
        return summary


    def _severity_to_log_level(self, severity: ErrorSeverity) -> int:
        """Convert error severity to logging level."""
        mapping = {
            ErrorSeverity.WARNING: logging.WARNING,
            ErrorSeverity.ERROR: logging.ERROR,
            ErrorSeverity.CRITICAL: logging.CRITICAL
        }
        return mapping[severity]
```

Milestone Checkpoints

After Milestone 1 (Block Elements):

- Run: `python -c "from markdown_renderer import parse_to_html; print(parse_to_html('# Hello\n\nWorld'))"`
- Expected: `<h1>Hello</h1>\n<p>World</p>`
- Verify: AST contains HEADING and PARAGRAPH nodes with correct hierarchy

After Milestone 2 (Inline Elements):

- Run: `python -c "from markdown_renderer import parse_to_html; print(parse_to_html('**bold** and *italic*'))"`
- Expected: `<p><strong>bold</strong> and <em>italic</em></p>`

- Verify: Inline nodes are properly nested within block nodes

After Milestone 3 (Lists):

- Run: `python -c "from markdown_renderer import parse_to_html; print(parse_to_html('- Item 1\n - Nested'))"`
- Expected: `<ul><li>Item 1<ul><li>Nested</li></ul></li></ul>`
- Verify: Nested list structures render with proper HTML nesting

After Milestone 4 (HTML Generation):

- Run full integration test with complex document containing all element types
- Expected: Valid HTML5 that passes W3C validation
- Verify: All special characters properly escaped, no malformed tags

Error Handling and Edge Cases

> **Milestone(s):** Milestone 1: Block Elements, Milestone 2: Inline Elements, Milestone 3: Lists, Milestone 4: HTML Generation

The Graceful Degradation Mental Model

Think of error handling in markdown parsing like being a helpful teacher grading a student's essay written in a second language. When you encounter grammatical errors or unclear sentences, you don't throw the entire essay in the trash — instead, you make reasonable interpretations, provide the best possible understanding of what the student meant, and continue reading. Similarly, when your markdown parser encounters malformed syntax or ambiguous input, it should make sensible assumptions, produce the most reasonable HTML output possible, and continue processing the rest of the document.

This graceful degradation approach is fundamental to markdown parsing because markdown exists in a messy, real-world context. Users type markdown in text editors without syntax highlighting, copy-paste content from various sources, and often mix valid markdown with informal text patterns. A parser that fails completely on the first syntax error would be virtually unusable. Instead, your parser should embody the principle of "be liberal in what you accept" — interpret ambiguous input charitably while still maintaining predictable behavior.

The key insight is that markdown parsing errors fall into two categories: recoverable inconsistencies where you can make reasonable assumptions about user intent, and structural corruption where continuing could produce misleading output. Your error handling strategy must distinguish between these cases and respond appropriately to each.

Parser Error Recovery

Parser error recovery represents the strategies your markdown renderer uses when it encounters input that doesn't conform to valid markdown syntax. The recovery approach differs significantly between the block parsing phase and the inline parsing phase because these phases have different error characteristics and different consequences for getting the interpretation wrong.

During block parsing, most errors involve ambiguous line classification or incomplete block structures. For example, a user might start a fenced code block with triple backticks but forget to close it, or they might create what looks like a heading but with invalid syntax. These errors are typically local to specific lines or line sequences, making recovery relatively straightforward. The parser can make reasonable assumptions about user intent and continue processing subsequent lines.

During inline parsing, errors more commonly involve mismatched delimiters or ambiguous emphasis markers. A user might write `**bold text without closing` or create nested emphasis that violates CommonMark precedence rules. These errors affect text spans within blocks, and recovery involves deciding how to interpret the malformed delimiters while preserving as much of the intended formatting as possible.

Block Parser Error Recovery Strategy

The block parser uses a **continuation-based recovery** approach where errors in one block don't prevent parsing of subsequent blocks. When the parser encounters malformed block syntax, it applies a hierarchy of fallback interpretations designed to preserve user intent whenever possible.

The recovery strategy follows this decision sequence:

1. **Attempt completion**: If the current block is incomplete (like an unclosed fenced code block), attempt to complete it using document boundaries or other structural clues
2. **Graceful degradation**: If completion isn't possible, degrade the block to a simpler, valid form (like converting a malformed heading to a paragraph)
3. **Content preservation**: Ensure that all text content appears somewhere in the output, even if the formatting is lost
4. **Error isolation**: Prevent errors in one block from affecting the interpretation of subsequent blocks
5. **Resume normal parsing**: Continue with the standard parsing algorithm for the next line sequence

For example, consider an unclosed fenced code block where the user writes:

````
Some paragraph text.

```python
def hello():
    print("world")

More paragraph text continues here.
````

The parser detects that the code block starting with triple backticks never encounters a closing fence. Rather than treating the entire rest of the document as code content, the recovery strategy recognizes that the user likely intended to close the code block before the next paragraph. The parser closes the code block implicitly at the next blank line followed by non-indented text, preserving both the code formatting and the subsequent paragraph content.

The block parser maintains an `ErrorCollector` that records recovery decisions without interrupting the parsing process. This allows the parser to complete successfully while still providing feedback about ambiguous input interpretation.
````

Error Type	Detection Method	Recovery Strategy	Fallback Behavior
Unclosed fenced code	End of document reached while in fenced code state	Close code block at document end	Preserve all content as code
Malformed ATX heading	Hash characters without following space or text	Convert to paragraph	Treat hash characters as literal text
Invalid Setext underline	Underline length doesn't match heading text	Treat as separate paragraph	Keep both lines as paragraph content
Incomplete blockquote	Lines start with `>` but have inconsistent spacing	Normalize spacing and continue	Remove `>` markers and treat as paragraph
Broken horizontal rule	Line has some dashes/asterisks but not enough	Convert to paragraph	Preserve characters as literal text
Mixed indentation in code	Some lines indented with spaces, others with tabs	Normalize to consistent indentation	Convert tabs to spaces using standard width

Inline Parser Error Recovery Strategy

The inline parser uses a **delimiter balancing recovery** approach that attempts to create the most reasonable interpretation of mismatched or ambiguous formatting markers. Because inline formatting can be nested and overlapping, the recovery strategy must maintain consistency with CommonMark's emphasis precedence rules while handling cases those rules don't cover.

The delimiter stack mechanism provides the foundation for error recovery. When the parser completes processing all text in a block but still has unmatched openers in the `DelimiterStack`, it applies recovery rules to determine how to handle the orphaned delimiters.

The recovery process follows this sequence:

1. **Attempt delayed matching**: Look for potential closers that were initially rejected due to precedence rules
2. **Pair partial matches**: Match delimiters even if they don't follow strict flanking rules, prioritizing user intent over specification compliance
3. **Convert to literal text**: Transform unmatched delimiters into regular text characters
4. **Preserve content**: Ensure all text content appears in the output with as much formatting as can be reasonably inferred
5. **Maintain consistency**: Apply the same recovery rules consistently across similar situations

For example, consider the text `**bold text with *italic inside` where the user forgot to close both the italic and bold formatting. The parser's delimiter stack contains an unmatched asterisk (for italic) and an unmatched double-asterisk (for bold). The recovery strategy recognizes that the user likely intended some formatting and converts the text to `**bold text with *italic inside**` by treating the entire span as bold text containing a literal asterisk character.

The inline parser also handles **intraword underscore** issues by implementing recovery rules that distinguish between intended emphasis and underscores that happen to appear within words. When the parser encounters underscores in positions where they could technically match according to basic delimiter rules but violate the intraword prohibition, it converts them to literal characters rather than attempting to create emphasis spans.

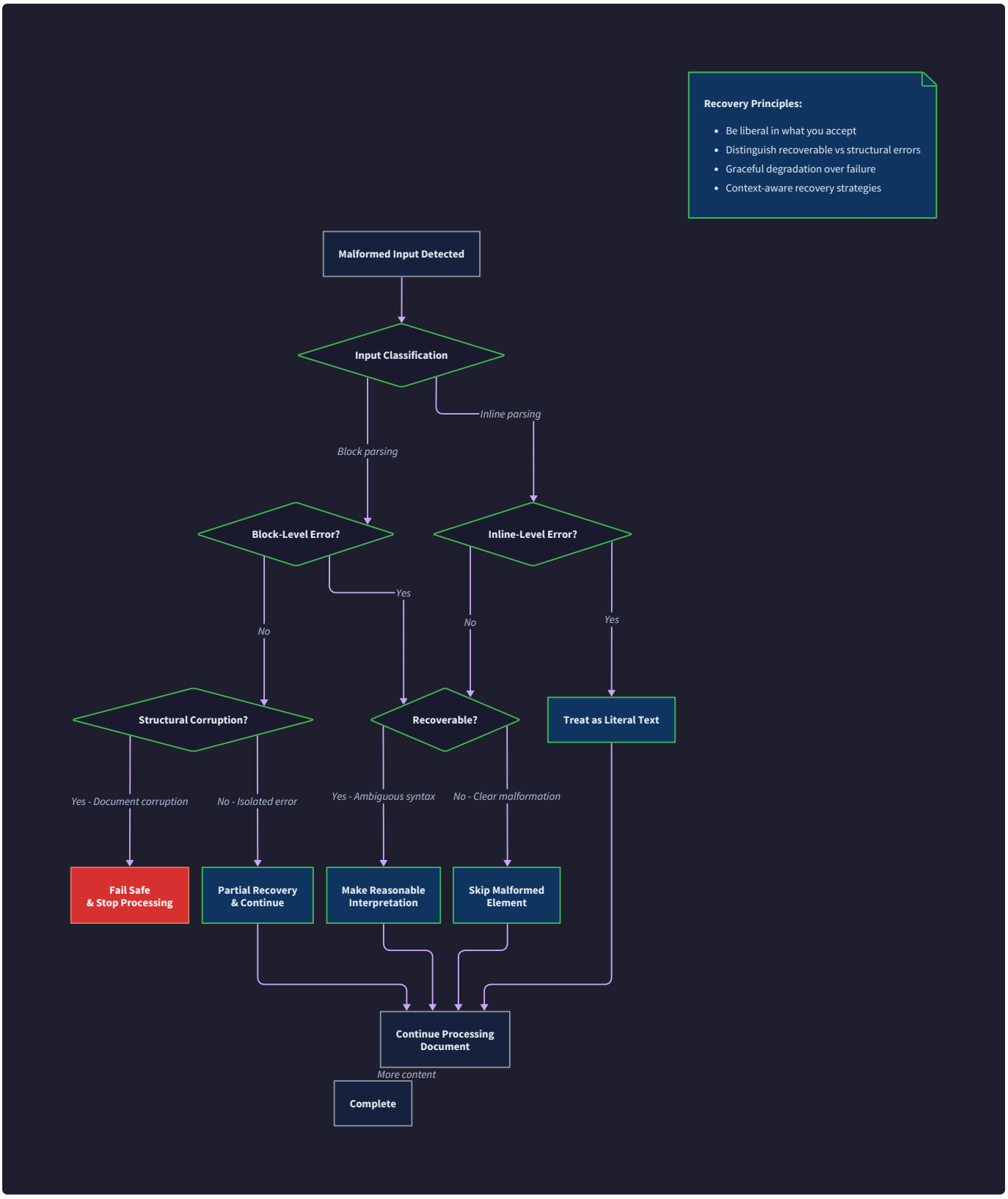
Error Type	Detection Method	Recovery Strategy	Fallback Behavior
Unmatched emphasis opener	Delimiter stack not empty after processing text	Convert to literal characters	Remove delimiter properties, keep as text
Mismatched emphasis types	Asterisk opener with underscore closer	Match delimiters of same type only	Leave unmatched delimiters as literals
Invalid nesting	Strong inside emphasis when emphasis can't contain strong	Allow nesting but adjust precedence	Follow CommonMark precedence rules
Broken link syntax	Opening bracket without closing bracket or parentheses	Treat brackets as literal text	Preserve all text including brackets
Incomplete image syntax	Exclamation mark with broken link syntax	Remove exclamation, apply link recovery	Fall back to link parsing rules
Unclosed inline code	Backtick without matching closing backtick	Close at end of current block	Treat remaining text as code content

Edge Case Handling

Edge cases in markdown parsing represent input patterns that are technically valid according to the specification but create ambiguous or counterintuitive results. Unlike errors, which involve clearly malformed syntax, edge cases require the parser to choose between multiple valid interpretations or handle boundary conditions that don't have obvious correct answers.

The markdown specification itself contains numerous edge cases because it attempts to codify the behavior of existing markdown implementations that evolved organically. Many of these edge cases arise from the interaction between different markdown features or from the need to handle whitespace, line breaks, and character sequences that have special meaning in some contexts but not others.

Your parser's edge case handling strategy should prioritize **consistent behavior** and **predictable results** over trying to guess user intent. When faced with ambiguous input, the parser should always make the same interpretation choice, even if that choice might not match what the user intended in every specific case.



Whitespace and Line Break Edge Cases

Whitespace handling represents one of the most complex categories of edge cases because markdown gives semantic meaning to certain whitespace patterns while treating others as insignificant. The parser must distinguish between whitespace that affects document structure, whitespace that affects formatting, and whitespace that should be preserved in the output.

Trailing whitespace on lines creates edge cases because two or more trailing spaces at the end of a line create a hard line break in the output, but a single trailing space is typically insignificant. However, users often have editors that automatically trim

trailing whitespace, or they copy-paste content that loses trailing spaces. The parser must decide whether to preserve these line breaks strictly according to the input or to apply some normalization.

The recommended approach is to implement **semantic whitespace preservation** where the parser preserves trailing spaces that clearly indicate intentional line breaks (two or more spaces) but normalizes accidental single trailing spaces. This requires the parser to track the original whitespace patterns during the preprocessing phase and apply consistent rules during HTML generation.

Mixed line endings create edge cases when documents contain a mixture of Unix (`\n`), Windows (`\r\n`), and legacy Mac (`\r`) line endings. While the `normalize_line_endings` function converts everything to Unix format, the parser must handle cases where the mixed line endings create unexpected blank lines or affect block boundary detection.

Tab expansion creates edge cases in indentation-sensitive contexts like code blocks and lists. The CommonMark specification requires tabs to be expanded to the next multiple of 4 spaces, but this can create counterintuitive results when users mix tabs and spaces or use different tab width assumptions. The `expand_tabs_to_spaces` function must handle these cases consistently while preserving the visual alignment that users intended.

Edge Case	Input Pattern	Expected Behavior	Implementation Strategy
Single trailing space	`Line content \n`	Ignore trailing space	Normalize during preprocessing
Double trailing space	`Line content \n`	Create hard line break	Preserve and convert to ` ` tag
Mixed tabs and spaces	`\t some text`	Normalize to consistent spacing	Expand tabs first, then calculate indentation
Empty lines with spaces	`\n`	Treat as blank line	Trim whitespace before blank line detection
CRLF followed by LF	`text\r\n\nmore`	Single paragraph break	Normalize line endings before processing
Unicode whitespace	`text\u00A0more`	Preserve non-breaking space	Only normalize ASCII whitespace characters

Delimiter Precedence Edge Cases

Delimiter precedence edge cases arise when multiple emphasis markers compete for the same text spans or when nested formatting creates ambiguous parsing situations. The CommonMark specification defines precedence rules for these cases, but implementing them correctly requires careful attention to delimiter matching order and flanking detection.

Overlapping emphasis spans occur when delimiter pairs would create emphasis spans that partially overlap rather than being properly nested. For example, the text `*emphasis **strong* text**` contains asterisks that would create overlapping emphasis if matched naively. The parser must apply precedence rules that ensure valid HTML nesting while producing predictable results.

The precedence algorithm processes delimiters in a specific order: shorter emphasis spans (single asterisks/underscores) are resolved before longer ones (double asterisks/underscores), and delimiters are matched from left to right within each precedence level. When conflicts arise, the earlier delimiter takes precedence and later conflicting delimiters are treated as literal text.

Intraword underscore handling creates edge cases when underscores appear within words in contexts where they could technically form valid emphasis according to basic flanking rules. The specification prohibits intraword underscore emphasis,

but determining what constitutes a "word" requires understanding Unicode character classes and handling edge cases with punctuation, numbers, and non-Latin scripts.

The `can_open_emphasis` and `can_close_emphasis` functions implement flanking detection that considers the characters immediately before and after each delimiter. For underscores, additional logic checks whether both the delimiter and its potential match are surrounded by word characters, which would violate the intraword prohibition.

Emphasis delimiter length matching creates edge cases when the parser encounters sequences of emphasis markers with different lengths. For example, `***text***` could be interpreted as strong emphasis (`**text**`) with an extra literal asterisk, or as emphasis (`*text*`) with extra literal asterisks, or as a combination of both emphasis and strong emphasis. The specification defines rules for these cases, but implementing them requires careful tracking of available delimiter lengths.

Edge Case	Input Pattern	CommonMark Interpretation	Implementation Notes
Overlapping emphasis	`*em **strong* text**`	`<em>em **strong</em> text**`	Earlier delimiter wins, later becomes literal
Intraword underscore	`snake_case_variable`	`snake_case_variable` (no emphasis)	Check word character boundaries for underscores
Triple asterisk	`***text***`	`<em><strong>text</strong></em>`	Parse as nested emphasis and strong
Mixed delimiter types	`*emphasis with __strong* text__`	`<em>emphasis with __strong</em> text__`	Don't match asterisks with underscores
Adjacent delimiters	`** **bold** **`	`** <strong>bold</strong> **`	Middle pair matches, outer delimiters are literals
Escaped delimiters	`\*not emphasis\*`	`*not emphasis*`	Process escapes before delimiter detection

List Structure Edge Cases

List structure edge cases arise from the complexity of list continuation rules, indentation handling, and the interaction between lists and other block elements. These edge cases often involve determining whether a line continues an existing list item, starts a new list item, or ends the list entirely.

Lazy continuation represents one of the most subtle edge cases in list parsing. The CommonMark specification allows certain types of content to continue list items without requiring the full indentation that would normally be needed. This means a line that appears to be a regular paragraph might actually be a continuation of a preceding list item, depending on context.

The lazy continuation rules apply differently to different types of content. Paragraph content can be lazily continued, but other block types like headings, code blocks, and blockquotes cannot. This creates edge cases where the same indentation level might continue a list in some contexts but not others.

Blank lines in lists affect whether lists are considered "tight" or "loose," which changes the HTML output by determining whether list item content is wrapped in paragraph tags. A single blank line within a list can change the formatting of the entire list, creating edge cases where small whitespace changes have large effects on the output.

List marker changes create edge cases when a sequence of list items uses inconsistent markers. For ordered lists, different numbering patterns or punctuation styles might indicate separate lists. For unordered lists, mixing different bullet characters (`-`, `*`, `+`) should create separate lists according to the specification, but users often mix them unintentionally.

The `ListContext` and `ListContextStack` classes handle these edge cases by tracking the specific marker types and indentation patterns for each nested list level. When the parser encounters a potential list item that doesn't match the current context, it must decide whether to end the current list and start a new one, or treat the line as non-list content.

Edge Case	Input Pattern	Expected Behavior	Implementation Strategy
Lazy continuation	List item followed by unindented text	Continue list item if paragraph content	Check content type and previous context
Blank line in tight list	List items with single blank line between	Convert entire list to loose formatting	Track blank line presence across all items
Mixed ordered markers	`1. item` followed by `1) item`	Create separate lists	Compare marker punctuation styles
Mixed unordered markers	`- item` followed by `* item`	Create separate lists	Compare bullet character types
Insufficient continuation indent	List item with under-indented continuation	End list, start new paragraph	Calculate required indentation precisely
List followed by code block	Indented code after list without blank line	Continue list vs. start code block	Require blank line before code block

Link and Image Edge Cases

Link and image parsing creates edge cases around URL validation, title extraction, and the interaction between link syntax and other markdown features. These edge cases often involve determining whether character sequences that look like links should be interpreted as links or treated as literal text.

Nested brackets in link text create parsing challenges because the link syntax uses brackets to delimit the link text, but the link text itself might contain brackets for other purposes. The parser must track bracket nesting levels and determine which closing bracket corresponds to the link text versus brackets that are part of the content.

URL validation creates edge cases because the parser must decide which character sequences constitute valid URLs without implementing a full URL parser. The specification allows a wide range of URL formats, including relative URLs, fragment identifiers, and URLs with unusual schemes. The parser must balance accepting valid URLs against avoiding false positives where parentheses or other punctuation accidentally close link syntax.

Link titles with quotes create edge cases when the title text contains the same quote characters used to delimit the title. The parser must handle escaped quotes within titles and determine which quotes are delimiters versus content.

The `LinkImageParser` class handles these edge cases through careful delimiter matching and context tracking. The parser maintains separate state for bracket nesting, parenthesis matching, and quote handling to ensure that complex link syntax is parsed correctly even when it contains characters that have special meaning in other contexts.

Edge Case	Input Pattern	Expected Behavior	Implementation Strategy
Nested brackets in text	`[link [with brackets]](url)`	Include brackets in link text	Track bracket nesting depth
URL with parentheses	`[text](url(with)parens)`	Include parentheses in URL	Balance parentheses in URL
Link title with quotes	`[text](url "title with \"quotes\"")`	Handle escaped quotes in title	Process escapes within quoted strings
Image vs link precedence	`[![image](img.png)](link.html)`	Image inside link	Parse inner image first, then outer link
Malformed URL	`[text](not a url)`	Treat as literal text	Validate basic URL structure
Missing link URL	`[text]()`	Create link with empty href	Allow empty URLs but create valid HTML

> **Key Insight**: Edge case handling is not about guessing user intent — it's about providing consistent, predictable behavior that users can learn and rely on. When faced with ambiguous input, always choose the interpretation that produces valid HTML and maintains consistency with the CommonMark specification.

Implementation Guidance

The error handling and edge case management system requires careful coordination between all parsing phases to ensure consistent behavior and comprehensive error recovery. The implementation focuses on building robust error collection and recovery mechanisms that don't interrupt the parsing pipeline while providing detailed information about how ambiguous input was interpreted.

Technology Recommendations

Component	Simple Option	Advanced Option
Error Collection	List of error messages with line numbers	Structured error objects with severity levels and recovery suggestions
Recovery Strategy	Fixed fallback rules hardcoded in parser	Configurable recovery policies with user-defined handlers
Edge Case Testing	Manual test cases for known edge cases	Property-based testing with random input generation
Validation	Basic HTML structure validation	Full CommonMark specification compliance testing

Recommended File Structure

The error handling system integrates across all parsing components:

```
markdown_renderer/
  src/
    parser/
      error_handling.py      ← ErrorCollector, ParseError classes
      recovery_strategies.py ← Recovery algorithms for each parser phase
      edge_cases.py          ← Edge case detection and handling utilities
    block_parser/
      block_parser.py        ← Modified to use ErrorCollector
      block_recovery.py      ← Block-specific recovery strategies
    inline_parser/
      inline_parser.py       ← Modified to use ErrorCollector
      delimiter_recovery.py  ← Delimiter matching recovery
    list_parser/
      list_parser.py         ← Modified to use ErrorCollector
      list_recovery.py       ← List structure recovery
    html_generator/
      html_generator.py      ← Modified to validate output
      validation.py          ← HTML structure validation utilities
    tests/
      error_cases/           ← Test files for error scenarios
      edge_cases/            ← Test files for edge case handling
      recovery_tests.py      ← Tests for recovery behavior
```

Error Collection Infrastructure

The error handling system begins with a comprehensive error collection mechanism that captures parsing issues without interrupting the parsing process. This infrastructure provides the foundation for both immediate recovery and post-processing analysis of parsing decisions.

PYTHON

```python
from enum import Enum

from dataclasses import dataclass

from typing import List, Optional, Any, Dict

import re


class ErrorSeverity(Enum):

    """Severity levels for parsing errors and edge cases."""

    WARNING = "warning"      # Ambiguous input with reasonable interpretation

    ERROR = "error"          # Invalid syntax with fallback behavior

    CRITICAL = "critical"    # Structural problems affecting output quality


@dataclass

class ParseError:

    """Structured representation of parsing errors and recovery decisions."""

    line_number: int

    column: Optional[int]

    severity: ErrorSeverity

    error_type: str

    message: str

    original_text: str

    recovery_action: str

    suggested_fix: Optional[str] = None

    context: Dict[str, Any] = None



    def __post_init__(self):

        if self.context is None:

            self.context = {}


class ErrorCollector:

    """Collects and manages parsing errors during markdown processing."""



    def __init__(self):
```

```python
        self.errors: List[ParseError] = []
        self.max_errors: int = 100  # Prevent excessive error collection
        self.collect_warnings: bool = True

    def add_error(self, line_number: int, error_type: str, message: str,
                  original_text: str, recovery_action: str,
                  severity: ErrorSeverity = ErrorSeverity.ERROR,
                  column: Optional[int] = None,
                  suggested_fix: Optional[str] = None,
                  **context) -> None:
        """Add a parsing error with recovery information."""
        # TODO 1: Check if max_errors limit reached, skip if exceeded
        # TODO 2: Create ParseError instance with all provided parameters
        # TODO 3: Set context dictionary from **context kwargs
        # TODO 4: Append error to self.errors list
        # TODO 5: If severity is WARNING and collect_warnings is False, skip adding
        pass

    def has_errors(self, min_severity: ErrorSeverity = ErrorSeverity.ERROR) -> bool:
        """Check if any errors of specified severity or higher exist."""
        # TODO 1: Iterate through self.errors
        # TODO 2: Compare each error's severity to min_severity
        # TODO 3: Return True if any error meets severity threshold
        pass

    def get_errors_by_line(self, line_number: int) -> List[ParseError]:
        """Get all errors that occurred on a specific line."""
        # TODO 1: Filter self.errors where error.line_number == line_number
        # TODO 2: Return filtered list
        pass
```

```
    def format_error_report(self) -> str:
        """Generate human-readable error report."""
        # TODO 1: Group errors by severity level
        # TODO 2: Format each error with line number, message, and recovery action
        # TODO 3: Include suggested fixes when available
        # TODO 4: Return formatted report string
        pass
```

Block Parser Recovery Implementation

The block parser recovery system handles incomplete or malformed block structures by maintaining recovery state alongside the normal parsing state machine. This allows the parser to detect error conditions and apply appropriate fallback strategies without losing document content.

```python
from typing import Optional, List, Tuple
from dataclasses import dataclass
from enum import Enum

class RecoveryStrategy(Enum):
    """Recovery strategies for different types of block parsing errors."""
    COMPLETE_BLOCK = "complete_block"      # Try to complete incomplete block
    CONVERT_TO_PARAGRAPH = "convert_to_paragraph"  # Fallback to paragraph
    PRESERVE_CONTENT = "preserve_content"   # Keep content, lose formatting
    SKIP_MALFORMED = "skip_malformed"      # Skip line, continue parsing

@dataclass
class BlockRecoveryContext:
    """Context information for block-level error recovery."""
    current_block_type: Optional[BlockType]
    lines_in_block: int
    expected_closing: Optional[str]
    fallback_strategy: RecoveryStrategy


def recover_incomplete_fenced_code(block_node: BlockNode,
                                   error_collector: ErrorCollector,
                                   line_number: int) -> None:
    """Recover from unclosed fenced code block at end of document."""
    # TODO 1: Add error to collector about unclosed fenced code block
    # TODO 2: Set block's closing fence implicitly
    # TODO 3: Mark block as complete in block attributes
    # TODO 4: Log recovery action taken
    pass

def recover_malformed_atx_heading(line: str, line_number: int,
                                  error_collector: ErrorCollector) -> BlockNode:
    """Recover from invalid ATX heading syntax by converting to paragraph."""
```

```
    # TODO 1: Detect specific malformation (no space after #, too many #, etc.)
    # TODO 2: Add error to collector with specific issue description
    # TODO 3: Create paragraph block node with original line content
    # TODO 4: Set recovery information in block attributes
    # TODO 5: Return paragraph block node
    pass

def recover_inconsistent_blockquote(lines: List[str], line_numbers: List[int],
                                    error_collector: ErrorCollector) -> BlockNode:
    """Recover from blockquote with inconsistent > prefixes."""
    # TODO 1: Identify lines with missing or malformed > prefixes
    # TODO 2: Add warning for each inconsistent line
    # TODO 3: Normalize all lines to have consistent > prefix
    # TODO 4: Create blockquote block with normalized content
    # TODO 5: Record normalization actions in block attributes
    pass
```

Inline Parser Recovery Implementation

The inline parser recovery system focuses on delimiter matching issues and emphasis precedence conflicts. The recovery strategies maintain the delimiter stack integrity while providing reasonable interpretations of malformed inline formatting.

```python
from typing import List, Optional, Tuple, Dict
from dataclasses import dataclass

@dataclass
class DelimiterRecoveryInfo:
    """Information about delimiter recovery actions."""
    original_position: int
    delimiter_char: str
    recovery_action: str
    matched_with: Optional[int] = None
    converted_to_literal: bool = False

def recover_unmatched_delimiters(delimiter_stack: DelimiterStack,
                                 text: str,
                                 error_collector: ErrorCollector) -> List[DelimiterRecoveryInfo]:
    """Recover from unmatched emphasis delimiters in delimiter stack."""
    # TODO 1: Iterate through remaining delimiters in stack
    # TODO 2: For each delimiter, determine if partial matching is possible
    # TODO 3: Apply recovery strategy (convert to literal, force match, etc.)
    # TODO 4: Add error/warning to collector for each recovery action
    # TODO 5: Create DelimiterRecoveryInfo for each action taken
    # TODO 6: Clear delimiter stack after recovery
    # TODO 7: Return list of recovery actions taken
    pass

def handle_intraword_underscore_conflict(text: str, start_pos: int, end_pos: int,
                                         error_collector: ErrorCollector) -> str:
    """Handle underscore emphasis that would violate intraword rules."""
    # TODO 1: Extract the text span between start_pos and end_pos
    # TODO 2: Check if both delimiter positions are within words
    # TODO 3: If intraword violation detected, add warning to collector
    # TODO 4: Convert delimiter characters to literal underscores
```

```
    # TODO 5: Return text with delimiters converted to literals
    pass


def recover_broken_link_syntax(text: str, bracket_pos: int,
                               error_collector: ErrorCollector) -> Tuple[str, bool]:
    """Recover from malformed link syntax by treating as literal text."""
    # TODO 1: Analyze link syntax starting at bracket_pos
    # TODO 2: Identify specific malformation (missing closing bracket, invalid URL, etc.)
    # TODO 3: Add error to collector with specific issue
    # TODO 4: Convert link syntax characters to literal text
    # TODO 5: Return tuple of (recovered_text, recovery_applied)
    pass
```

Edge Case Detection and Handling

The edge case handling system provides consistent behavior for ambiguous input patterns that are technically valid but could be interpreted multiple ways. The implementation focuses on predictable rule application rather than trying to guess user intent.

```python
import re

from typing import List, Tuple, Optional, Dict, Any


# Edge case detection patterns

MIXED_LINE_ENDING_PATTERN = re.compile(r'\r\n|\r|\n')

TRAILING_WHITESPACE_PATTERN = re.compile(r'[ \t]+$', re.MULTILINE)

UNICODE_WHITESPACE_PATTERN = re.compile(r'[\u00A0\u1680\u2000-
\u200B\u2028\u2029\u202F\u205F\u3000\uFEFF]')


def normalize_edge_case_whitespace(text: str,

                                   preserve_semantic_breaks: bool = True) -> Tuple[str, List[str]]:

    """Normalize whitespace while handling edge cases consistently."""

    # TODO 1: Track normalization actions taken for reporting

    # TODO 2: Convert all line endings to Unix format (\n)

    # TODO 3: Handle trailing whitespace - preserve semantic line breaks (2+ spaces)

    # TODO 4: Expand tabs to spaces using 4-space tab width

    # TODO 5: Normalize Unicode whitespace to ASCII equivalents where appropriate

    # TODO 6: Remove trailing whitespace from blank lines

    # TODO 7: Return tuple of (normalized_text, list_of_actions_taken)

    pass


def detect_emphasis_precedence_conflict(delimiters: List[Delimiter]) -> List[Tuple[int, int, str]]:

    """Detect potential emphasis delimiter precedence conflicts."""

    # TODO 1: Sort delimiters by position

    # TODO 2: Check for overlapping emphasis spans

    # TODO 3: Identify conflicts between asterisk and underscore delimiters

    # TODO 4: Find cases where delimiter length precedence applies

    # TODO 5: Return list of (start_pos, end_pos, conflict_type) tuples

    pass


def handle_list_marker_inconsistency(current_marker: str, new_marker: str,

                                     context: Dict[str, Any]) -> Tuple[bool, str]:

    """Handle inconsistent list markers according to CommonMark rules."""
```

```
    # TODO 1: Parse marker types (ordered vs unordered, specific characters)

    # TODO 2: Check if markers are compatible (same list vs separate lists)

    # TODO 3: For ordered lists, check numbering continuity

    # TODO 4: For unordered lists, check bullet character consistency

    # TODO 5: Return tuple of (should_continue_list, explanation)

    pass
```

Comprehensive Error Testing Framework

The error handling system requires extensive testing to ensure consistent behavior across all edge cases and recovery scenarios. The testing framework validates both the recovery actions taken and the quality of the final output.

```python
import pytest

from typing import List, Dict, Any, Optional

from dataclasses import dataclass


@dataclass

class ErrorTestCase:

    """Test case for error handling and recovery behavior."""

    name: str

    markdown_input: str

    expected_html: str

    expected_errors: List[Dict[str, Any]]

    recovery_actions: List[str]

    edge_case_type: str


def create_error_test_cases() -> List[ErrorTestCase]:

    """Create comprehensive test cases for error handling scenarios."""

    return [

        ErrorTestCase(

            name="unclosed_fenced_code_block",

            markdown_input="```python\ndef test():\n    pass\n\nMore text here.",

            expected_html="<pre><code class=\"language-python\">def test():\n    pass\n</code>
</pre>\n<p>More text here.</p>",

            expected_errors=[{

                "error_type": "unclosed_fenced_code",

                "severity": "ERROR",

                "recovery_action": "closed_at_next_paragraph"

            }],

            recovery_actions=["close_code_block_at_paragraph_break"],

            edge_case_type="block_structure"

        ),

        # TODO: Add test cases for:

        # - Malformed ATX headings
```

```
        # - Unmatched emphasis delimiters
        # - Intraword underscore conflicts
        # - Broken link syntax
        # - Mixed list markers
        # - Whitespace edge cases
        # - Nested delimiter conflicts
        # - Unicode edge cases
    ]

def test_error_recovery_consistency():
    """Test that error recovery produces consistent results."""
    # TODO 1: Load error test cases
    # TODO 2: For each test case, parse markdown with error collection enabled
    # TODO 3: Verify that expected errors were collected
    # TODO 4: Verify that recovery actions match expectations
    # TODO 5: Verify that HTML output matches expected result
    # TODO 6: Test that parsing same input twice produces identical results
    pass

def test_edge_case_predictability():
    """Test that edge cases produce predictable, consistent behavior."""
    # TODO 1: Create variations of edge case inputs
    # TODO 2: Parse each variation multiple times
    # TODO 3: Verify identical output across all parsing attempts
    # TODO 4: Verify that similar edge cases produce similar handling
    # TODO 5: Test that edge case handling follows CommonMark specification
    pass
```

Debugging Error Handling Issues

The error handling system includes comprehensive debugging support to help identify why specific recovery actions were taken and how to adjust the behavior if needed.

Symptom	Likely Cause	How to Diagnose	Fix Approach
Parser silently ignores malformed input	Recovery strategy too aggressive	Enable error collection, check error report	Adjust recovery strategy to be more conservative
Parser fails on minor syntax issues	Recovery strategy too strict	Check ErrorCollector for excessive ERROR level issues	Add more WARNING-level recoveries for minor issues
Inconsistent output for similar inputs	Edge case handling not standardized	Test with variations of problematic input	Implement consistent rule application
Valid markdown parsed incorrectly	Edge case detector too aggressive	Check for false positive edge case detection	Refine detection patterns to be more specific
HTML output contains formatting errors	Recovery created invalid nesting	Validate HTML structure after recovery	Add HTML validation step after recovery actions
Error messages not helpful	ParseError lacks context information	Check ParseError.context field population	Add more contextual information during error creation

The error handling and edge case management system provides the foundation for a robust markdown parser that handles real-world input gracefully while maintaining predictable behavior. The combination of structured error collection, consistent recovery strategies, and comprehensive edge case handling ensures that users receive useful output even when their markdown input contains syntax errors or ambiguous patterns.

Testing Strategy

> **Milestone(s):** Milestone 1: Block Elements, Milestone 2: Inline Elements, Milestone 3: Lists, Milestone 4: HTML Generation

The Quality Assurance Mental Model

Think of testing a markdown renderer like quality control in a translation service. You have documents coming in one language (markdown) and going out in another (HTML), and you need to verify that the meaning, structure, and formatting are preserved accurately across this transformation. Just as a translation service would test with simple phrases, complex literary works, edge cases like idioms, and even malformed input, your markdown renderer needs systematic verification at every level.

The testing strategy serves as both a safety net and a learning accelerator. Each test category validates different properties of correctness, from basic functionality to edge case resilience. The systematic approach ensures that as you build each milestone, you can confidently rely on previously implemented components while adding new functionality.

Test Categories and Properties

Testing a markdown renderer requires validating multiple dimensions of correctness simultaneously. Unlike simple algorithmic problems where you test input-output pairs, document transformation involves structural correctness, semantic preservation, and format compliance. Each test category validates specific properties and catches different classes of errors.

Unit Test Categories

Block Parser Unit Tests verify that individual block-level elements are recognized and parsed correctly in isolation. These tests focus on the `BlockParser` component and validate that each block type handler correctly identifies, extracts, and structures content.

Test Category	Property Validated	Example Test Case	Expected Behavior
ATX Heading Recognition	Heading level detection accuracy	`### Hello World`	Creates `BlockNode` with `block_type=HEADING`, level 3, content "Hello World"
Paragraph Boundary Detection	Text grouping correctness	Two lines separated by blank line	Creates two separate `BlockNode` instances with `block_type=PARAGRAPH`
Fenced Code Block Parsing	Content preservation with metadata	Triple backticks with language hint	Preserves exact whitespace, extracts language identifier
Blockquote Nesting	Hierarchical structure building	Multiple `>` levels	Creates nested `BlockNode` tree with proper parent-child relationships
Horizontal Rule Recognition	Pattern matching precision	Various `---`, `***`, `___` patterns	Distinguishes valid rules from similar-looking text

Inline Parser Unit Tests validate that formatting elements within text are correctly identified, paired, and nested. These tests exercise the `DelimiterStack` and emphasis matching algorithms.

Test Category	Property Validated	Example Test Case	Expected Behavior
Emphasis Delimiter Matching	Proper opener-closer pairing	`**bold** and *italic*`	Matches delimiters correctly, doesn't cross-pair
Intraword Underscore Handling	Context-sensitive formatting	`snake_case_variable`	Doesn't apply emphasis to underscores within words
Nested Formatting	Hierarchical inline structure	`**bold with *italic* inside**`	Creates proper AST nesting without conflicts
Link Syntax Parsing	URL and title extraction	`[text](url "title")`	Extracts all components with proper escaping
Escape Sequence Processing	Literal character handling	`\*not emphasis\*`	Converts escape sequences to literal characters

List Parser Unit Tests verify the complex indentation tracking and nesting logic that handles one of markdown's most challenging parsing scenarios.

Test Category	Property Validated	Example Test Case	Expected Behavior
Marker Type Consistency	List type enforcement	Mixed `-` and `*` markers	Treats as separate lists per CommonMark rules
Indentation Level Tracking	Nesting depth calculation	Various indentation patterns	Builds correct hierarchical structure
Lazy Continuation	Content flow rules	Multi-line items without markers	Properly continues items without requiring markers
Tight vs Loose Lists	Spacing semantics	Lists with/without blank lines	Sets appropriate formatting flags for HTML generation
List Item Content Parsing	Multi-paragraph item handling	Items containing multiple blocks	Nests paragraphs, code blocks within list items

Integration Test Categories

Parser Pipeline Integration Tests validate that the complete parsing pipeline produces coherent results when components work together. These tests catch interaction bugs that unit tests miss.

Test Category	Property Validated	Example Test Case	Expected Behavior
Block-Inline Coordination	Proper processing order	Paragraph with emphasis	Block parser creates paragraph, inline parser processes emphasis within
List-Inline Integration	Complex nested parsing	List items with links and emphasis	Maintains both list structure and inline formatting
Escape Context Preservation	Consistent escape handling	Escapes in different contexts	Escapes work consistently across block and inline contexts
AST Structure Integrity	Tree validity	Complex nested documents	Produces well-formed AST with proper parent-child links
State Machine Transitions	Parser state consistency	Mixed block types	State transitions don't leave parser in inconsistent state

HTML Generation Integration Tests ensure that the parsed AST converts to valid, semantically correct HTML that preserves the original document's intent.

Test Category	Property Validated	Example Test Case	Expected Behavior
HTML Validity	Standards compliance	Complex nested structures	Produces HTML5-compliant output that passes validation
Character Escaping	Security and correctness	Special characters in various contexts	Properly escapes content vs attribute contexts
Semantic Preservation	Meaning conservation	Complex formatting combinations	HTML conveys same semantic meaning as original markdown
Pretty Printing	Output readability	Nested block structures	Produces properly indented, human-readable HTML
Custom Renderer Integration	Extensibility verification	Documents using custom renderers	Plugin system works without breaking core functionality

Property-Based Test Categories

Property-based tests validate system behaviors that should hold regardless of specific input content. These tests are particularly valuable for catching edge cases and ensuring robust error handling.

Property Category	Invariant Tested	Test Approach	Validation Method
Roundtrip Stability	`parse(markdown).to_text() ≈ original`	Generate varied markdown inputs	Compare semantic equivalence
Parsing Determinism	Same input always produces same AST	Repeat parsing with identical input	Verify AST structure matches exactly
Error Recovery Bounds	Malformed input produces valid output	Inject syntax errors systematically	Check output is still valid HTML
Memory Safety	No resource leaks during parsing	Parse large documents repeatedly	Monitor memory usage patterns
Performance Linearity	Parse time scales reasonably	Vary document size systematically	Verify no exponential behavior

> The key insight in testing document transformation is that you're validating not just correctness, but also consistency across different contexts. A delimiter that works in paragraphs must also work in list items, blockquotes, and other nested contexts.

Error Handling Test Categories

Error handling tests ensure graceful degradation when encountering malformed or edge-case input. These tests validate the `ErrorCollector` and recovery mechanisms described in the Error Handling section.

Error Category	Failure Mode	Test Case	Expected Recovery
Unclosed Fenced Code	EOF before closing fence	Code block without `'''` end	Convert to indented code block
Malformed Links	Invalid URL or missing parts	`[text](invalid`	Render as literal text
Unmatched Emphasis	Mismatched delimiters	`**bold *italic**`	Handle precedence correctly
Invalid List Structure	Inconsistent indentation	Mixed tabs and spaces	Normalize to consistent spacing
Mixed Line Endings	Platform inconsistencies	Mix of `\n`, `\r\n`, `\r`	Normalize to consistent format

Architecture Decision: Test Organization Strategy

> **Decision: Hierarchical Test Organization with Component Isolation**
>
> - **Context**: Need to balance comprehensive coverage with maintainable test suite organization, while supporting incremental milestone development
> - **Options Considered**:
>   1. Single large test file with all cases
>   2. Test files mirroring source code structure exactly
>   3. Hierarchical organization by test type and milestone
> - **Decision**: Hierarchical organization with separate directories for unit, integration, and property-based tests
> - **Rationale**: Allows learners to focus on relevant tests during each milestone while maintaining clear separation between test categories. Supports both incremental development and comprehensive validation.
> - **Consequences**: Requires more initial setup but provides better long-term maintainability and learning progression

Milestone Verification Checkpoints

Each milestone represents a significant capability milestone that should be thoroughly verified before proceeding. The verification checkpoints provide concrete, measurable criteria for determining when a milestone is complete and the foundation is solid for the next phase.

Milestone 1: Block Elements Verification

Functional Verification Criteria for the block parsing infrastructure ensure that the foundational parsing capabilities work correctly across all supported block types.

Block Type	Verification Test	Input Example	Expected Output Structure
ATX Headings	All heading levels parse correctly	`# H1\n## H2\n### H3\n#### H4\n##### H5\n###### H6`	Six `BlockNode` instances with correct levels 1-6
Setext Headings	Underline styles recognized	`Heading 1\n========\nHeading 2\n--------`	Two `BlockNode` instances with levels 1 and 2
Paragraphs	Text grouping with blank line separation	`Para 1\n\nPara 2\n\nPara 3`	Three separate `BlockNode` instances with `block_type=PARAGRAPH`
Fenced Code	Language hints and content preservation	```` ```python\nprint("hello")\n``` ````	`BlockNode` with language="python", exact content preservation
Indented Code	Four-space rule compliance	`code line 1\n code line 2`	Single code `BlockNode` with preserved indentation
Blockquotes	Simple and nested structures	`> Quote\n>> Nested\n> Back to level 1`	Nested `BlockNode` tree with proper hierarchy
Horizontal Rules	Various marker patterns	`---\n***\n___`	Three `BlockNode` instances with `block_type=HORIZONTAL_RULE`

State Machine Verification ensures that the `BlockParserState` transitions work correctly and don't leave the parser in inconsistent states.

State Transition	Test Scenario	Expected Behavior	Error Detection
`LOOKING_FOR_BLOCK` → `IN_PARAGRAPH`	Regular text line encountered	Creates new paragraph block, transitions state	Should not remain in looking state
`IN_PARAGRAPH` → `LOOKING_FOR_BLOCK`	Blank line encountered	Finalizes paragraph, resets state	Paragraph should be added to document
`IN_FENCED_CODE` → `LOOKING_FOR_BLOCK`	Matching closing fence	Finalizes code block with preserved content	Missing closure should be detected
Mixed block transitions	Complex document with all block types	All blocks parsed correctly in sequence	No state machine confusion

Manual Verification Steps provide concrete actions to verify milestone completion:

1. **Create comprehensive test document** containing all supported block types in various combinations
2. **Run block parser** on the test document and examine the generated AST structure
3. **Verify AST node count** matches expected number of blocks (use `debug_print_ast` function)
4. **Check block type assignment** - each block should have correct `block_type` value
5. **Validate content extraction** - block content should match original text with proper whitespace handling
6. **Test edge cases** - empty blocks, blocks at document boundaries, maximum nesting levels

Milestone 1 Checkpoint Command Sequence:

```bash
# Run comprehensive block parser tests
python -m pytest tests/unit/block_parser/ -v

# Run milestone 1 integration tests
python -m pytest tests/integration/milestone1/ -v

# Manual verification with debug output
python -c "
from markdown_parser import MarkdownParser
parser = MarkdownParser()
ast = parser.parse_blocks(open('test_documents/milestone1_comprehensive.md').read())
parser.debug_print_ast(ast, 0)
"
```

Milestone 2: Inline Elements Verification

Delimiter Matching Verification ensures that the `DelimiterStack` correctly handles all emphasis and formatting patterns.

Formatting Type	Test Pattern	Expected Result	Common Failure Mode
Bold (asterisk)	`**bold text**`	`InlineNode` with `inline_type=STRONG`	Fails to match across line breaks
Bold (underscore)	`__bold text__`	`InlineNode` with `inline_type=STRONG`	Incorrectly matches intraword
Italic (asterisk)	`*italic text*`	`InlineNode` with `inline_type=EMPHASIS`	Conflicts with list markers
Italic (underscore)	`_italic text_`	`InlineNode` with `inline_type=EMPHASIS`	Triggers on snake_case variables
Nested emphasis	`**bold with *italic* inside**`	Properly nested `InlineNode` tree	Incorrect precedence resolution
Inline code	`` `code span` ``	`InlineNode` with `inline_type=CODE`	Doesn't preserve internal formatting
Links	`[text](url)`	`InlineNode` with URL and text extracted	Fails on nested brackets
Images	`![alt](url)`	`InlineNode` with alt text and URL	Confuses with link syntax

Flanking Detection Verification validates the complex rules for when delimiters can open or close emphasis spans.

Flanking Scenario	Test Case	Expected Behavior	Rule Applied
Left-flanking asterisk	`*emphasis` at word start	Can open emphasis	Preceded by whitespace or punctuation
Right-flanking asterisk	`emphasis*` at word end	Can close emphasis	Followed by whitespace or punctuation
Non-flanking underscore	`snake_case_var`	Cannot open/close	Surrounded by alphanumeric characters
Punctuation flanking	`"*quoted emphasis*"`	Can open and close	Punctuation counts as whitespace
Mixed flanking	`*emphasis* and **bold**`	Correct delimiter pairing	Doesn't cross-match different types

Manual Verification Steps for inline parsing:

1. **Create inline formatting test suite** with all supported inline elements
2. **Verify delimiter stack behavior** by adding debug output to delimiter matching
3. **Test precedence rules** with complex nested formatting combinations
4. **Validate escape sequence handling** - escaped delimiters should render literally
5. **Check URL parsing accuracy** for links and images with various URL formats

Milestone 3: Lists Verification

List Structure Verification validates the complex indentation tracking and nesting logic implemented in the `ListContext` system.

List Type	Test Pattern	Expected Structure	Nesting Validation
Simple unordered	`- Item 1\n- Item 2`	Single `BlockNode` list with two items	Flat structure, no nesting
Simple ordered	`1. First\n2. Second`	Single ordered list with sequential numbering	Preserves start numbers
Nested unordered	`- Parent\n - Child\n- Sibling`	Nested list structure	Child list within first item
Mixed nesting	`1. Ordered\n - Unordered child\n2. Next`	Mixed list types in hierarchy	Proper type preservation
Multi-paragraph items	`- Para 1\n\n Para 2\n- Next item`	Items containing multiple blocks	Paragraph nesting within items

Indentation Tracking Verification ensures that the `ListContext` and indentation calculation work correctly across different scenarios.

Indentation Scenario	Test Case	Expected Behavior	Error Detection
Consistent spaces	All items use 2-space indent	Proper nesting levels calculated	Should build clean hierarchy
Tab vs space mixing	Some items use tabs, others spaces	Normalized to consistent spacing	Should detect and handle gracefully
Lazy continuation	Multi-line items without full indent	Content correctly attributed to items	Should not break item boundaries
Over-indentation	Items indented more than required	Treated as code blocks within items	Should preserve extra indentation

Manual Verification Steps for list parsing:

1. **Test complex nesting scenarios** with multiple levels and mixed types
2. **Verify tight vs loose list detection** by checking spacing between items
3. **Validate continuation line handling** for items spanning multiple lines
4. **Test list boundary detection** - where lists end and other blocks begin
5. **Check marker consistency enforcement** within single lists

Milestone 4: HTML Generation Verification

HTML Validity Verification ensures that the generated HTML is standards-compliant and semantically correct.

Validity Aspect	Verification Method	Expected Result	Validation Tool
HTML5 Compliance	W3C validator on generated output	Zero validation errors	Nu HTML Checker
Character Escaping	Special characters in various contexts	Proper entity encoding	Manual inspection + validation
Tag Nesting	Complex nested structures	Properly closed and nested tags	HTML parser verification
Attribute Formatting	Links, images with various attributes	Quoted, escaped attribute values	Attribute syntax validation
Semantic Correctness	Markdown meaning preserved in HTML	Equivalent semantic meaning	Manual comparison

Pretty Printing Verification validates that the optional formatting produces readable, well-structured HTML output.

Formatting Aspect	Test Case	Expected Output	Quality Measure
Consistent Indentation	Nested block structures	Each nesting level properly indented	2-space increments
Line Breaking	Long documents	Appropriate line breaks between blocks	Readable structure
Inline Element Handling	Mixed block and inline content	Inline elements on same line as parents	No unnecessary breaks
Whitespace Preservation	Code blocks and preformatted text	Exact whitespace preservation where needed	Content fidelity

End-to-End Integration Verification tests the complete pipeline from markdown input to HTML output.

Integration Test	Input Document Type	Validation Criteria	Success Metric
CommonMark Specification Examples	Official CommonMark test suite	Matches expected HTML output	100% spec compliance
Real-world Documents	GitHub README files, documentation	Produces usable, readable HTML	Visual inspection passes
Stress Testing	Large documents (1000+ lines)	Completes without errors or crashes	Performance within bounds
Malformed Input Recovery	Documents with syntax errors	Produces valid HTML despite errors	Graceful degradation

Final Verification Command Sequence:

```bash
# Run complete test suite
python -m pytest tests/ -v --cov=markdown_parser

# Validate against CommonMark spec examples
python scripts/run_spec_tests.py

# Generate HTML from sample documents
python -c "
from markdown_parser import MarkdownParser
parser = MarkdownParser()
html = parser.parse_to_html(open('samples/comprehensive_test.md').read())
print(html)
" > output.html

# Validate generated HTML
curl -X POST -F "file=@output.html" https://validator.w3.org/nu/

# Performance baseline test
time python -c "
from markdown_parser import MarkdownParser
parser = MarkdownParser()
html = parser.parse_to_html(open('samples/large_document.md').read())
"
```

> The verification checkpoints serve as both quality gates and confidence builders. Each checkpoint confirms that the implementation is solid before adding complexity in the next milestone. This systematic approach prevents the common trap of building on unstable foundations.

Implementation Guidance

This section provides concrete tools and approaches for implementing a comprehensive testing strategy that supports both learning and quality assurance throughout the markdown renderer development process.

Technology Recommendations Table

Test Category	Simple Option	Advanced Option
Unit Testing Framework	`unittest` (Python standard library)	`pytest` with fixtures and parametrization
Test Data Management	Inline strings in test methods	External test files with YAML/JSON metadata
Property-Based Testing	Manual edge case enumeration	`hypothesis` library for automated input generation
HTML Validation	Manual inspection with browser	Automated W3C validator API integration
Performance Testing	Simple timing with `time` module	`pytest-benchmark` with statistical analysis
Coverage Analysis	Visual code inspection	`coverage.py` with branch coverage reporting
Test Organization	Single test directory	Structured hierarchy with milestone-based organization

Recommended File Structure

The testing infrastructure should mirror the component architecture while supporting milestone-based development progression:

```
markdown_parser/
├── tests/
│   ├── __init__.py
│   ├── conftest.py                    ← pytest configuration and shared fixtures
│   ├── test_data/                     ← shared test documents and expected outputs
│   │   ├── milestone1_blocks.md       ← comprehensive block parsing test cases
│   │   ├── milestone2_inline.md       ← inline formatting test cases
│   │   ├── milestone3_lists.md        ← list parsing test cases
│   │   ├── milestone4_complete.md     ← full integration test document
│   │   ├── expected_outputs/          ← expected HTML outputs for integration tests
│   │   └── malformed_inputs/          ← error handling test cases
│   ├── unit/                          ← component-level tests
│   │   ├── __init__.py
│   │   ├── test_preprocessor.py       ← line processing and normalization tests
│   │   ├── test_block_parser.py       ← block parsing unit tests
│   │   ├── test_inline_parser.py      ← inline parsing unit tests
│   │   ├── test_list_parser.py        ← list parsing unit tests
│   │   ├── test_html_generator.py     ← HTML generation unit tests
│   │   └── test_ast_nodes.py          ← AST data structure tests
│   ├── integration/                   ← multi-component interaction tests
│   │   ├── __init__.py
│   │   ├── test_milestone1.py         ← block parsing integration tests
│   │   ├── test_milestone2.py         ← inline parsing integration tests
│   │   ├── test_milestone3.py         ← list parsing integration tests
│   │   ├── test_milestone4.py         ← complete pipeline integration tests
│   │   └── test_pipeline.py           ← end-to-end pipeline tests
│   ├── property/                      ← property-based and generative tests
│   │   ├── __init__.py
│   │   ├── test_roundtrip.py          ← roundtrip stability tests
│   │   ├── test_performance.py        ← performance and scalability tests
│   │   └── test_error_recovery.py     ← error handling property tests
│   ├── spec_compliance/               ← CommonMark specification compliance
│   │   ├── __init__.py
│   │   ├── test_commonmark_spec.py    ← official spec test runner
│   │   └── commonmark_examples.json   ← spec examples in JSON format
│   └── manual/                        ← manual testing scripts and tools
│       ├── debug_ast_viewer.py        ← interactive AST exploration tool
│       ├── html_validator.py          ← HTML validation utility
│       └── performance_profiler.py    ← performance analysis tool
├── scripts/                           ← testing automation scripts
│   ├── run_milestone_tests.py         ← milestone-specific test runner
│   ├── generate_coverage_report.py    ← coverage analysis automation
│   └── validate_html_output.py        ← batch HTML validation
```

Infrastructure Starter Code

Complete Test Configuration Setup (`tests/conftest.py`):

```python
"""
Pytest configuration and shared test fixtures for markdown parser testing.
Provides common test utilities and data that can be reused across test modules.
"""

import pytest
from pathlib import Path
from typing import List, Dict, Any
import json
import yaml

from markdown_parser.core import MarkdownParser
from markdown_parser.ast_nodes import ASTNode, BlockNode, InlineNode
from markdown_parser.html_generator import HtmlRenderer

# Test data directory path
TEST_DATA_DIR = Path(__file__).parent / "test_data"

@pytest.fixture
def parser():
    """Provides a fresh MarkdownParser instance for each test."""
    return MarkdownParser()

@pytest.fixture
def html_renderer():
    """Provides an HtmlRenderer instance with default settings."""
    return HtmlRenderer(pretty_print=True)

@pytest.fixture
def sample_documents():
    """Loads all sample documents from test_data directory."""
    documents = {}
    for md_file in TEST_DATA_DIR.glob("*.md"):
        with open(md_file, 'r', encoding='utf-8') as f:
```

```
            documents[md_file.stem] = f.read()
    return documents

@pytest.fixture
def expected_outputs():
    """Loads expected HTML outputs for integration testing."""
    outputs = {}
    expected_dir = TEST_DATA_DIR / "expected_outputs"
    for html_file in expected_dir.glob("*.html"):
        with open(html_file, 'r', encoding='utf-8') as f:
            outputs[html_file.stem] = f.read().strip()
    return outputs

@pytest.fixture
def commonmark_examples():
    """Loads CommonMark specification examples for compliance testing."""
    spec_file = TEST_DATA_DIR.parent / "spec_compliance" / "commonmark_examples.json"
    if spec_file.exists():
        with open(spec_file, 'r', encoding='utf-8') as f:
            return json.load(f)
    return []

class ASTTestHelpers:
    """Helper methods for testing AST structure and content."""


    @staticmethod
    def count_nodes_by_type(root: ASTNode, node_type: str) -> int:
        """Count all nodes of specified type in AST tree."""
        count = 0
        if hasattr(root, 'node_type') and root.node_type.name == node_type:
            count += 1
        for child in getattr(root, 'children', []):
```

```python
            count += ASTTestHelpers.count_nodes_by_type(child, node_type)
        return count

    @staticmethod
    def extract_text_content(node: ASTNode) -> str:
        """Extract all text content from node and descendants."""
        text_parts = []
        if hasattr(node, 'text_content') and node.text_content:
            text_parts.append(node.text_content)
        if hasattr(node, 'inline_content') and node.inline_content:
            text_parts.append(node.inline_content)
        for child in getattr(node, 'children', []):
            text_parts.append(ASTTestHelpers.extract_text_content(child))
        return ''.join(text_parts)

    @staticmethod
    def find_nodes_with_content(root: ASTNode, content_substring: str) -> List[ASTNode]:
        """Find all nodes containing specified text content."""
        matches = []
        node_text = ASTTestHelpers.extract_text_content(root)
        if content_substring in node_text:
            matches.append(root)
        for child in getattr(root, 'children', []):
            matches.extend(ASTTestHelpers.find_nodes_with_content(child, content_substring))
        return matches

@pytest.fixture
def ast_helpers():
    """Provides AST testing helper methods."""
    return ASTTestHelpers()

class HTMLValidationHelpers:
```

```
    """Helper methods for validating generated HTML output."""


    @staticmethod
    def validate_html_structure(html: str) -> Dict[str, Any]:
        """Basic HTML structure validation."""
        from html.parser import HTMLParser


        class ValidationParser(HTMLParser):
            def __init__(self):
                super().__init__()
                self.tag_stack = []
                self.errors = []
                self.warnings = []


            def handle_starttag(self, tag, attrs):
                if tag not in ['br', 'hr', 'img', 'input', 'meta', 'link']:
                    self.tag_stack.append(tag)


            def handle_endtag(self, tag):
                if self.tag_stack and self.tag_stack[-1] == tag:
                    self.tag_stack.pop()
                else:
                    self.errors.append(f"Mismatched closing tag: {tag}")


            def error(self, message):
                self.errors.append(message)


        parser = ValidationParser()
        try:
            parser.feed(html)
            return {
```

```
                'valid': len(parser.errors) == 0,
                'errors': parser.errors,
                'warnings': parser.warnings,
                'unclosed_tags': parser.tag_stack
            }
        except Exception as e:
            return {
                'valid': False,
                'errors': [f"Parse error: {str(e)}"],
                'warnings': [],
                'unclosed_tags': []
            }

    @staticmethod
    def check_character_escaping(html: str) -> Dict[str, bool]:
        """Verify that special characters are properly escaped."""
        import re

        # Check for unescaped special characters in content (not attributes)
        content_pattern = r'>([^<]*)<'
        content_matches = re.findall(content_pattern, html)

        results = {
            'ampersands_escaped': all('&' not in content or '&amp;' in content or '&lt;' in content or
'&gt;' in content or '&quot;' in content for content in content_matches),
            'less_than_escaped': all('<' not in content for content in content_matches),
            'greater_than_escaped': all('>' not in content for content in content_matches),
            'quotes_handled': True  # More complex check needed for attribute contexts
        }

        return results
```

```python
@pytest.fixture
def html_helpers():
    """Provides HTML validation helper methods."""
    return HTMLValidationHelpers()

# Performance testing utilities
@pytest.fixture
def performance_tracker():
    """Provides performance tracking utilities for tests."""
    import time
    import psutil
    import os

    class PerformanceTracker:
        def __init__(self):
            self.process = psutil.Process(os.getpid())
            self.start_time = None
            self.start_memory = None

        def start_tracking(self):
            self.start_time = time.perf_counter()
            self.start_memory = self.process.memory_info().rss

        def stop_tracking(self):
            end_time = time.perf_counter()
            end_memory = self.process.memory_info().rss

            return {
                'duration_seconds': end_time - self.start_time,
                'memory_delta_bytes': end_memory - self.start_memory,
                'final_memory_mb': end_memory / (1024 * 1024)
```

```
            }


    return PerformanceTracker()
```

Complete Test Data Generator (`scripts/generate_test_cases.py`):

```python
"""
Generates comprehensive test cases for all markdown parser components.
Creates both positive test cases and negative/edge case scenarios.
"""

import os
import json
from pathlib import Path
from typing import List, Dict, Any, Tuple

def generate_block_test_cases() -> Dict[str, str]:
    """Generate comprehensive block parsing test cases."""

    test_cases = {
        'milestone1_blocks': """# Top Level Heading

This is a regular paragraph with some text that spans
multiple lines but should be grouped together.
```

Second Level Heading

Another paragraph here.

Third Level Heading

```
This is indented code block
with multiple lines
and preserved spacing
```

```python
# This is fenced code block

def hello_world():

    print("Hello, World!")
```

```
Plain fenced code without language
```

> This is a blockquote that spans multiple lines and maintains formatting

> ## Heading in blockquote
>
> Paragraph in blockquote with blank lines around it.

Horizontal rule above.

Another horizontal rule.

Third horizontal rule style.

Nested Blockquotes

> Level 1 quote
>
> > Level 2 nested quote
> > Still level 2 Back to level 1

Mixed Block Types

Regular paragraph followed by:

```
Code block
```

> Quote block

More regular text.

Edge Cases

Heading with extra spaces

Heading with leading spaces

#No space after hash

Empty heading content above.

####### Invalid - too many hashes

Setext Heading Level 1

Setext Heading Level 2

Not a setext heading ---regular text continues--- """,

`    'edge_cases_blocks': """`
Document starting with blank lines.

Heading immediately after blanks

Paragraph with trailing spaces.

`Indented code with trailing spaces`
`Second line of code`

```javascript
// Fenced code with empty lines


console.log("test");
```

> Blockquote with trailing spaces
> Second line

Text immediately after horizontal rule.

Mixed line endings test case would be handled by test runner.

Last paragraph without trailing newline.""" }

`return test_cases`
def generate_inline_test_cases() -> Dict[str, str]: """Generate comprehensive inline parsing test cases."""

`test_cases = {`
`    'milestone2_inline': """# Inline Formatting Tests`

Basic Emphasis

This paragraph contains **bold text** and *italic text* for testing.

Also testing **bold with underscores** and *italic with underscores*.

Nested Formatting

Here is **bold text with *italic inside* it** for testing nesting.

And here is *italic text with **bold inside** it* for reverse nesting.

Code Spans

Here is `inline code` within a sentence.

Here is `code with **bold** inside` where the bold should not render.

Links and Images

This is a simple link in text.

This is a link with title including title.

Here is an image:

 Alt text

And an image with title:

 Alt text

Complex Link Cases

Link with *emphasis* in text

Link with `code` in text

Edge Cases for Emphasis

This has *emphasis at start* of sentence.

This has emphasis *at the end*.

Emphasis at paragraph start looks like this.

Words with under_scores_inside should not be emphasized.

But *this should be emphasized* because it has word boundaries.

Escape Sequences

This has *escaped asterisks* that should not format.

And \`escaped backticks\\` that should not format.

Also \[escaped brackets\] and \! exclamation points.

Complex Combinations

Bold text with a link inside.

Italic text with `code` and link inside.

Link with **bold** and *italic* text

Delimiter Matching Edge Cases

This is bold but **this** is also **bold**.

Single and *multiple* and *italic* spans.

**Unmatched bold at end

*Unmatched italic at end

Bold with italic inside and back to normal.

Mismatched delimiters: **bold* should not work.

URL Edge Cases

[Link with spaces in URL](http://example.com/path with spaces)

Link with special chars

http://autolink.example.com

autolink@email.com """,

```
    'delimiter_edge_cases': """**bold**normal*italic*normal**bold**
```

bold italic combined

bold *italic in bold* bold

*italic **bold in italic** italic*

snake_case_variable_name

emphasis_with_underscores

double__underscore__test

emphasis and **bold** and `code`

link*italic*link

 image

bold

 image

```
code with *emphasis* inside
```

`code` and *emphasis* and `code`

bold across lines

italic across
lines

This is **bold and link and `code`** together.

This is italic with

 *image*

inside.

Edge case: ** (empty emphasis)

Edge case: __ (empty emphasis)

Edge case: ` (single backtick)

Edge case: ```

[](empty link)

 """ }

`return test_cases`
def generate_list_test_cases() -> Dict[str, str]: """Generate comprehensive list parsing test cases."""

`test_cases = {`
`    'milestone3_lists': """# List Parsing Tests`

Simple Unordered Lists

- First item
- Second item
- Third item

- Alternative bullet style
- Second item with asterisk
- Third item

- Plus sign bullets
- Second item with plus
- Third item

Simple Ordered Lists

1. First numbered item
2. Second numbered item
3. Third numbered item

1) Alternative parenthesis style
2) Second item with parenthesis
3) Third item

Nested Lists

- Top level item
  - Nested item level 2
  - Another nested item
    - Deeper nesting level 3
    - More deep content
  - Back to level 2
- Back to top level

Mixed List Types

1. Ordered parent
   - Unordered child
   - Another unordered child
     1. Ordered grandchild
     2. Another ordered grandchild
   - Back to unordered child level
2. Second ordered parent

Multi-paragraph List Items

- First item with single paragraph
- Second item with multiple paragraphs

  This is the second paragraph of the second item. It continues here.

  And this is a third paragraph in the same item.
- Third item back to single paragraph

Lists with Code Blocks

1. Item with indented code block:

   ```
   def example_function():
       return "Hello World"
   ```
2. Item with fenced code block:

```python
    def another_example():
        print("In a list!")
```

3. Regular item after code

Lists with Blockquotes

- Item with blockquote:

  > This is a quote inside a list item. It continues on multiple lines.

- Another regular item

Tight vs Loose Lists

Tight list (no blank lines):

- Item 1
- Item 2
- Item 3

Loose list (blank lines between items):

- Item 1
- Item 2
- Item 3

Complex Nesting with Mixed Content

1. First ordered item

   This item has a paragraph.

   - Nested unordered list
   - With multiple items

     1. And further nesting
     2. With ordered list

        > And even blockquotes Inside nested items

   Back to paragraph in first item.

2. Second ordered item with `inline code`

Edge Cases

- Item with *emphasis* and **bold** and links
- Item with `inline code` formatting
- Item ending with colon:
- Item with trailing spaces
  -Item without space after marker -Nested without proper spacing -Deep nesting spacing issues

Lazy Continuation

- This is a lazy continuation where the second line doesn't have proper indentation but should still be part of the same item.
  - Nested item

with lazy continuation that continues without indentation.

List Marker Consistency

- First item with dash
- Second item with dash

* This starts a new list with asterisk
* Because marker changed

1. Ordered list
2. Continues with same marker

1. This starts new list with different marker
2. Because marker style changed

Lists at Document Boundaries

First paragraph.

- List immediately after paragraph
- Without blank line

Last list at end of document:

- Item 1
- Item 2""",

  `'list_edge_cases': """1.No space after period`

2. Item with proper space

-No space after dash

- Item with proper space

1. Multiple spaces after period

- Multiple spaces after dash

1. Indented ordered list
2. Second indented item

- List with lots of content

  Multiple paragraphs in item.

  > Blockquote in item

  `Code in item`
  More text.

- Next item

15. List starting with large number

16. Continues correctly

17. List starting with zero

18. Next item

-1. Invalid negative number (should be paragraph)

1. Item
2. Invalid nesting (different marker continuation)

1.Item without space 2. Item with wrong indentation

Text

1. List after text without blank line
2. Should still work per CommonMark

Paragraph

- Unordered after paragraph
- Also works

Mixed indentation: 1. Tab indented 2. Space indented (may cause issues)

Empty items:

- Item after empty
-

Trailing content:

1. Item with trailing content...
2. ...continues here """ }

   return test_cases

def generate_html_output_test_cases() -> Dict[str, str]: """Generate expected HTML outputs for integration testing."""

`expected_outputs = {`
`    'simple_blocks': """<!DOCTYPE html>`

Main Heading

This is a simple paragraph with some text.

Subheading

Another paragraph here.

`Code block content`
`with multiple lines`

> This is a quote block.

""",
`    'inline_formatting': """<h1>Inline Tests</h1>`
This has **bold** and *italic* text.

Also `inline code` and links.

Complex: **bold with *italic* inside**.

""",
`    'nested_lists': """<ol>`
First item

- Nested item
- Another nested

Second item
""" }
`return expected_outputs`
def create_test_data_files(): """Create all test data files in the appropriate directory structure."""

`# Create directory structure`
`base_dir = Path("tests/test_data")`
`base_dir.mkdir(parents=True, exist_ok=True)`

`expected_dir = base_dir / "expected_outputs"`
`expected_dir.mkdir(exist_ok=True)`

`malformed_dir = base_dir / "malformed_inputs"`
`malformed_dir.mkdir(exist_ok=True)`

`# Generate and write block test cases`
`block_cases = generate_block_test_cases()`
`for name, content in block_cases.items():`

```
    with open(base_dir / f"{name}.md", 'w', encoding='utf-8') as f:
        f.write(content)

# Generate and write inline test cases
inline_cases = generate_inline_test_cases()
for name, content in inline_cases.items():
    with open(base_dir / f"{name}.md", 'w', encoding='utf-8') as f:
        f.write(content)

# Generate and write list test cases
list_cases = generate_list_test_cases()
for name, content in list_cases.items():
    with open(base_dir / f"{name}.md", 'w', encoding='utf-8') as f:
        f.write(content)

# Generate and write expected HTML outputs
html_outputs = generate_html_output_test_cases()
for name, content in html_outputs.items():
    with open(expected_dir / f"{name}.html", 'w', encoding='utf-8') as f:
        f.write(content)

print(f"Generated test data files in {base_dir}")
```

if **name** == "**main**": create_test_data_files()

Core Logic Skeleton Code

Block Parser Test Template (`tests/unit/test_block_parser.py`):

```python
"""
Unit tests for block-level element parsing.
Tests each block type in isolation to verify correct parsing behavior.
"""

import pytest
from markdown_parser.block_parser import BlockParser, BlockParserState
from markdown_parser.ast_nodes import BlockNode, NodeType, BlockType
from markdown_parser.preprocessor import LineInfo

class TestBlockParserCore:
    """Test core block parser functionality."""

    def test_parser_initialization(self):
        """Verify parser initializes with correct default state."""
        parser = BlockParser()
        # TODO: Assert parser state is LOOKING_FOR_BLOCK
        # TODO: Assert document root is created with correct type
        # TODO: Assert line buffer is empty

    def test_line_processing_basic(self, parser):
        """Test basic line processing without specific block types."""
        lines = [
            LineInfo("Regular text", 1, False, 0),
            LineInfo("", 2, True, 0),
            LineInfo("More text", 3, False, 0)
        ]

        # TODO: Process line sequence through parser
        # TODO: Verify correct number of blocks created
        # TODO: Check that blank lines separate blocks appropriately
        # TODO: Validate final parser state
```

```
class TestATXHeadings:
    """Test ATX-style heading parsing (#, ##, ###, etc.)."""

    @pytest.mark.parametrize("level,markdown,expected_text", [
        (1, "# Heading 1", "Heading 1"),
        (2, "## Heading 2", "Heading 2"),
        (3, "### Heading 3", "Heading 3"),
        (4, "#### Heading 4", "Heading 4"),
        (5, "##### Heading 5", "Heading 5"),
        (6, "###### Heading 6", "Heading 6"),
    ])
    def test_atx_heading_levels(self, parser, level, markdown, expected_text):
        """Test all valid ATX heading levels parse correctly."""
        line = LineInfo(markdown, 1, False, 0)

        # TODO: Process line through parser
        # TODO: Verify BlockNode created with block_type=HEADING
        # TODO: Check heading level attribute matches expected
        # TODO: Verify text content matches expected_text
        # TODO: Ensure no child nodes created for simple heading

    def test_atx_heading_with_trailing_hashes(self, parser):
        """Test ATX headings with optional trailing hash marks."""
        test_cases = [
            ("# Heading #", "Heading"),
            ("## Heading ##", "Heading"),
            ("### Heading ###", "Heading"),
            ("# Heading ####", "Heading"),  # Mismatched trailing
        ]
```

```
        for markdown, expected_text in test_cases:
            # TODO: Parse each test case
            # TODO: Verify trailing hashes are stripped correctly
            # TODO: Check that mismatched trailing hashes are handled

    def test_atx_heading_edge_cases(self, parser):
        """Test edge cases for ATX heading parsing."""
        edge_cases = [
            ("#No space", False),  # Should not be heading
            ("# ", True),          # Empty heading content
            ("#######", False),    # Too many hashes
            ("  # Indented", False), # Indented headings invalid
        ]

        for markdown, should_be_heading in edge_cases:
            # TODO: Process each edge case
            # TODO: Verify heading detection matches expected
            # TODO: For non-headings, ensure treated as paragraph
            # TODO: For invalid cases, check error recovery

class TestSetextHeadings:
    """Test Setext-style heading parsing (underlined headings)."""

    def test_setext_h1_detection(self, parser):
        """Test detection of H1 Setext headings with = underlines."""
        markdown_lines = [
            LineInfo("Main Heading", 1, False, 0),
            LineInfo("============", 2, False, 0)
        ]

        # TODO: Process both lines through parser
```

```
        # TODO: Verify single BlockNode created with HEADING type
        # TODO: Check heading level is 1
        # TODO: Verify text content is "Main Heading"
        # TODO: Ensure underline is not included in content

    def test_setext_h2_detection(self, parser):
        """Test detection of H2 Setext headings with - underlines."""
        # TODO: Similar test structure as H1 but with dashes
        # TODO: Verify heading level is 2
        # TODO: Test various dash lengths and patterns

    def test_setext_invalid_cases(self, parser):
        """Test cases that look like Setext but should not be headings."""
        invalid_cases = [
            # Blank line before underline breaks Setext
            ["Text", "", "==="],
            # Mixed underline characters
            ["Text", "=-="],
            # Underline without preceding text
            ["", "==="],
        ]

        # TODO: Test each invalid case
        # TODO: Verify they are treated as separate paragraphs/blocks
        # TODO: Check no heading nodes are created

class TestParagraphs:
    """Test paragraph detection and grouping."""

    def test_simple_paragraph(self, parser):
        """Test basic paragraph creation from consecutive lines."""
```

```
        lines = [
            LineInfo("First line of paragraph.", 1, False, 0),
            LineInfo("Second line continues paragraph.", 2, False, 0),
            LineInfo("Third line also continues.", 3, False, 0)
        ]

        # TODO: Process lines through parser
        # TODO: Verify single BlockNode created with PARAGRAPH type
        # TODO: Check all lines are included in paragraph content
        # TODO: Verify line breaks are preserved appropriately

    def test_paragraph_separation(self, parser):
        """Test that blank lines separate paragraphs correctly."""
        lines = [
            LineInfo("First paragraph.", 1, False, 0),
            LineInfo("", 2, True, 0),
            LineInfo("Second paragraph.", 3, False, 0),
            LineInfo("", 4, True, 0),
            LineInfo("Third paragraph.", 5, False, 0)
        ]

        # TODO: Process all lines
        # TODO: Verify three separate BlockNode instances created
        # TODO: Check each has PARAGRAPH type
        # TODO: Ensure content separation is correct
        # TODO: Verify no blank line content in paragraphs

class TestCodeBlocks:
    """Test both fenced and indented code block parsing."""

    def test_fenced_code_basic(self, parser):
```

```
        """Test basic fenced code block parsing."""
        lines = [
            LineInfo("```", 1, False, 0),
            LineInfo("code line 1", 2, False, 0),
            LineInfo("code line 2", 3, False, 0),
            LineInfo("```", 4, False, 0)
        ]

        # TODO: Process lines through parser
        # TODO: Verify BlockNode created with CODE_BLOCK type
        # TODO: Check exact content preservation including whitespace
        # TODO: Ensure fence markers are not included in content
        # TODO: Verify parser state transitions correctly

    def test_fenced_code_with_language(self, parser):
        """Test fenced code blocks with language identifiers."""
        lines = [
            LineInfo("```python", 1, False, 0),
            LineInfo("def hello():", 2, False, 0),
            LineInfo("    print('Hello')", 3, False, 0),
            LineInfo("```", 4, False, 0)
        ]

        # TODO: Parse and verify language attribute is set
        # TODO: Check that language info doesn't appear in content
        # TODO: Verify code content is preserved exactly

    def test_indented_code_block(self, parser):
        """Test indented code block parsing (4+ spaces)."""
        lines = [
            LineInfo("    code line 1", 1, False, 4),
```

```
            LineInfo("    code line 2", 2, False, 4),
            LineInfo("        more indented", 3, False, 8),
            LineInfo("    back to base", 4, False, 4)
        ]

        # TODO: Process indented lines
        # TODO: Verify CODE_BLOCK type created
        # TODO: Check that base indentation is removed
        # TODO: Ensure relative indentation is preserved
        # TODO: Test transition out of code block state

    def test_code_block_edge_cases(self, parser):
        """Test edge cases for code block parsing."""
        # TODO: Test unclosed fenced code blocks (EOF)
        # TODO: Test empty code blocks
        # TODO: Test mixed indentation in code blocks
        # TODO: Test fence markers inside code content
        # TODO: Test language identifiers with spaces/special chars

class TestBlockquotes:
    """Test blockquote parsing including nested blockquotes."""

    def test_simple_blockquote(self, parser):
        """Test basic blockquote parsing."""
        lines = [
            LineInfo("> This is a quote", 1, False, 0),
            LineInfo("> It continues here", 2, False, 0)
        ]

        # TODO: Process blockquote lines
        # TODO: Verify BLOCKQUOTE type BlockNode created
```

```
        # TODO: Check that > markers are stripped from content
        # TODO: Ensure content is preserved correctly

    def test_nested_blockquotes(self, parser):
        """Test nested blockquote structures."""
        lines = [
            LineInfo("> Level 1", 1, False, 0),
            LineInfo(">> Level 2", 2, False, 0),
            LineInfo(">>> Level 3", 3, False, 0),
            LineInfo(">> Back to Level 2", 4, False, 0),
            LineInfo("> Back to Level 1", 5, False, 0)
        ]

        # TODO: Parse nested structure
        # TODO: Verify proper nesting hierarchy in AST
        # TODO: Check parent-child relationships are correct
        # TODO: Ensure content at each level is accurate

class TestHorizontalRules:
    """Test horizontal rule parsing with different marker styles."""

    @pytest.mark.parametrize("marker_line", [
        "---",
        "***",
        "___",
        "----",
        "*****",
        "_____",
        "- - -",
        "* * *",
        "_ _ _"
```

```
    ])
    def test_horizontal_rule_variants(self, parser, marker_line):
        """Test all valid horizontal rule patterns."""
        line = LineInfo(marker_line, 1, False, 0)

        # TODO: Process horizontal rule line
        # TODO: Verify HORIZONTAL_RULE type BlockNode created
        # TODO: Check that content is empty or minimal
        # TODO: Ensure parser state resets correctly after rule

    def test_horizontal_rule_invalid(self, parser):
        """Test patterns that look like HR but should not be."""
        invalid_patterns = [
            "--",      # Too short
            "- -",     # Too short
            "-- -",    # Mixed patterns
            "***text", # Text after markers
        ]

        for pattern in invalid_patterns:
            # TODO: Process each invalid pattern
            # TODO: Verify NO horizontal rule is created
            # TODO: Check that line is treated as paragraph instead

# TODO: Add comprehensive integration tests that combine multiple block types
# TODO: Add state machine transition tests
# TODO: Add error recovery tests for malformed blocks
# TODO: Add performance tests with large documents
```

Milestone Checkpoint Implementation (`scripts/run_milestone_tests.py`):

```python
"""
Automated milestone verification script.
Runs appropriate test suites for each milestone and provides clear pass/fail feedback.
"""

import sys
import subprocess
import json
from pathlib import Path
from typing import Dict, List, Any, Optional
import time

class MilestoneTestRunner:
    """Coordinates milestone-specific testing and validation."""

    def __init__(self, project_root: Path):
        self.project_root = project_root
        self.test_results = {}

    def run_milestone_1_tests(self) -> Dict[str, Any]:
        """Run all tests required for Milestone 1: Block Elements."""
        print("🧪 Running Milestone 1: Block Elements Tests")

        results = {
            'milestone': 1,
            'description': 'Block Elements',
            'test_categories': {},
            'overall_status': 'pending'
        }

        # TODO: Run block parser unit tests
        # TODO: Command: pytest tests/unit/test_block_parser.py -v
```

```
        # TODO: Capture output and parse results
        # TODO: Check for specific test method completions

        # TODO: Run block parsing integration tests
        # TODO: Command: pytest tests/integration/test_milestone1.py -v
        # TODO: Verify AST structure correctness

        # TODO: Run manual verification steps
        # TODO: Parse comprehensive test document
        # TODO: Validate AST node counts and types
        # TODO: Check for proper block detection

        # TODO: Generate summary report
        # TODO: Set overall_status based on all test results
        # TODO: Return detailed results dictionary

        return results

    def run_milestone_2_tests(self) -> Dict[str, Any]:
        """Run all tests required for Milestone 2: Inline Elements."""
        print("🧪 Running Milestone 2: Inline Elements Tests")

        # TODO: Similar structure to milestone 1
        # TODO: Focus on inline parser unit tests
        # TODO: Test delimiter stack functionality
        # TODO: Verify emphasis and link parsing
        # TODO: Check escape sequence handling

        return {'milestone': 2, 'status': 'not_implemented'}

    def run_milestone_3_tests(self) -> Dict[str, Any]:
```

```
        """Run all tests required for Milestone 3: Lists."""
        print("🧪 Running Milestone 3: Lists Tests")

        # TODO: List parser unit tests
        # TODO: Indentation tracking verification
        # TODO: Nested list structure validation
        # TODO: Tight vs loose list detection

        return {'milestone': 3, 'status': 'not_implemented'}

    def run_milestone_4_tests(self) -> Dict[str, Any]:
        """Run all tests required for Milestone 4: HTML Generation."""
        print("🧪 Running Milestone 4: HTML Generation Tests")

        # TODO: HTML generator unit tests
        # TODO: End-to-end integration tests
        # TODO: HTML validity verification
        # TODO: Character escaping validation
        # TODO: Performance benchmarking

        return {'milestone': 4, 'status': 'not_implemented'}

    def verify_milestone_completion(self, milestone: int) -> bool:
        """Verify that a specific milestone meets all completion criteria."""
        verification_methods = {
            1: self._verify_milestone_1,
            2: self._verify_milestone_2,
            3: self._verify_milestone_3,
            4: self._verify_milestone_4
        }
```

```
        if milestone in verification_methods:
            return verification_methods[milestone]()
        else:
            print(f"❌ Unknown milestone: {milestone}")
            return False

    def _verify_milestone_1(self) -> bool:
        """Verify Milestone 1 completion criteria."""
        print("✅ Verifying Milestone 1 completion...")

        # TODO: Check that all required block types are implemented
        # TODO: Verify AST structure correctness
        # TODO: Test state machine transitions
        # TODO: Validate error handling for malformed blocks

        required_block_types = ['HEADING', 'PARAGRAPH', 'CODE_BLOCK', 'BLOCKQUOTE', 'HORIZONTAL_RULE']

        # TODO: For each block type:
        # TODO:   - Create test input
        # TODO:   - Parse through block parser
        # TODO:   - Verify correct BlockNode type created
        # TODO:   - Check content extraction accuracy

        return False  # TODO: Replace with actual verification logic

    def _verify_milestone_2(self) -> bool:
        """Verify Milestone 2 completion criteria."""
        # TODO: Test all inline formatting types
        # TODO: Verify delimiter stack functionality
        # TODO: Check emphasis precedence rules
        # TODO: Test link and image parsing
```

```
        return False

    def _verify_milestone_3(self) -> bool:
        """Verify Milestone 3 completion criteria."""
        # TODO: Test list parsing accuracy
        # TODO: Verify nesting level calculations
        # TODO: Check marker consistency enforcement
        # TODO: Test lazy continuation handling
        return False

    def _verify_milestone_4(self) -> bool:
        """Verify Milestone 4 completion criteria."""
        # TODO: Test complete pipeline functionality
        # TODO: Verify HTML validity
        # TODO: Check character escaping
        # TODO: Test performance requirements
        return False

    def run_comprehensive_test_suite(self) -> Dict[str, Any]:
        """Run all tests across all milestones."""
        print("🚀 Running Comprehensive Test Suite")

        all_results = {
            'timestamp': time.time(),
            'milestones': {},
            'overall_summary': {}
        }

        # Run each milestone's tests
        milestone_runners = [
            self.run_milestone_1_tests,
```

```
            self.run_milestone_2_tests,
            self.run_milestone_3_tests,
            self.run_milestone_4_tests
        ]

        for i, runner in enumerate(milestone_runners, 1):
            try:
                results = runner()
                all_results['milestones'][i] = results
                print(f"✅ Milestone {i}: {results.get('overall_status', 'unknown')}")
            except Exception as e:
                print(f"❌ Milestone {i} failed: {str(e)}")
                all_results['milestones'][i] = {
                    'milestone': i,
                    'overall_status': 'error',
                    'error': str(e)
                }

        # TODO: Calculate overall summary statistics
        # TODO: Identify which milestones are complete
        # TODO: Generate recommendations for next steps

        return all_results

    def generate_test_report(self, results: Dict[str, Any]) -> str:
        """Generate human-readable test report."""
        report_lines = [
            "# Markdown Renderer Test Report",
            f"Generated: {time.ctime(results['timestamp'])}",
            "",
            "## Milestone Summary",
```

```
        ""
    ]

    for milestone_num, milestone_results in results['milestones'].items():
        status = milestone_results.get('overall_status', 'unknown')
        description = milestone_results.get('description', f'Milestone {milestone_num}')

        status_emoji = {
            'passed': '✅',
            'failed': '❌',
            'partial': '⚠️',
            'not_implemented': '⏳',
            'error': '🔥'
        }.get(status, '❓')

        report_lines.extend([
            f"### {status_emoji} Milestone {milestone_num}: {description}",
            f"Status: {status.title()}",
            ""
        ])

        # TODO: Add detailed test category results
        # TODO: Include specific failure information
        # TODO: Add recommendations for fixing issues

    # TODO: Add overall recommendations section
    # TODO: Include next steps guidance
    # TODO: Add debugging tips for common failures

    return "\n".join(report_lines)
```

```python
def main():
    """Main entry point for milestone testing."""
    if len(sys.argv) < 2:
        print("Usage: python run_milestone_tests.py <milestone_number|all>")
        print("Example: python run_milestone_tests.py 1")
        print("Example: python run_milestone_tests.py all")
        sys.exit(1)

    project_root = Path(__file__).parent.parent
    runner = MilestoneTestRunner(project_root)

    target = sys.argv[1].lower()

    if target == 'all':
        results = runner.run_comprehensive_test_suite()
        report = runner.generate_test_report(results)
        print("\n" + "="*50)
        print(report)
    elif target.isdigit():
        milestone_num = int(target)
        if 1 <= milestone_num <= 4:
            success = runner.verify_milestone_completion(milestone_num)
            if success:
                print(f"🎉 Milestone {milestone_num} is complete!")
            else:
                print(f"❌ Milestone {milestone_num} needs more work.")
                sys
```

Debugging Guide

> **Milestone(s):** Milestone 1: Block Elements, Milestone 2: Inline Elements, Milestone 3: Lists, Milestone 4: HTML Generation

The Detective Work Mental Model

Think of debugging a markdown parser like being a detective at a crime scene where the "crime" is incorrect HTML output. You have evidence (the wrong output), witnesses (intermediate parsing states), and forensic tools (debugging utilities). Just as a detective systematically examines evidence and follows leads, debugging text parsing requires systematic examination of each transformation stage to identify where the parsing pipeline diverged from expected behavior.

The key insight is that text parsing bugs often manifest far from their source. A missing paragraph tag might be caused by incorrect line splitting in preprocessing, a malformed list structure might stem from indentation calculation errors, and broken emphasis formatting could result from delimiter stack corruption. Effective debugging requires understanding these cause-and-effect relationships and knowing where to look for root causes.

Symptom-Based Debugging Table

The following comprehensive table maps common symptoms learners encounter to their likely root causes, specific diagnostic approaches, and targeted fixes. This systematic approach helps developers identify and resolve issues efficiently rather than randomly trying different solutions.

Symptom	Likely Root Cause	Diagnostic Steps	Specific Fix
Headings render as plain paragraphs	ATX heading regex not matching or incorrect heading detection	1. Print raw line content before heading detection 2. Test `ATX_HEADING_PATTERN` regex against failing line 3. Check if line has unexpected characters or encoding issues	Fix regex pattern to handle edge cases like trailing spaces, or add line content normalization before pattern matching
Code blocks appear as regular paragraphs	Fenced code detection failing or state machine not transitioning	1. Print current `BlockParserState` when processing code fence lines 2. Verify fence marker counting (minimum 3 backticks) 3. Check if closing fence detection is working	Ensure `FENCED_CODE_START` pattern matches opening fence, verify state transitions to `IN_FENCED_CODE`, fix closing fence detection logic
Emphasis markers show as literal text	Delimiter stack not processing or flanking detection broken	1. Print delimiter stack contents after each character 2. Check flanking detection results for failing delimiters 3. Verify `can_open_emphasis` and `can_close_emphasis` logic	Fix flanking detection algorithm, ensure delimiters are properly added to stack, verify opener-closer matching logic
Nested emphasis produces malformed HTML	Incorrect emphasis delimiter precedence or double processing	1. Trace delimiter matching decisions step by step 2. Check if same text being processed multiple times 3. Verify delimiter precedence rules implementation	Implement proper emphasis precedence (** before *, length-based matching), ensure single-pass processing of inline content
Lists render as separate paragraphs	List marker detection failing or indentation calculation wrong	1. Print detected marker info for each line 2. Verify indentation level calculations 3. Check `ListMarkerInfo` extraction results	Fix marker detection regex patterns, correct indentation calculation (handle tabs vs spaces), ensure marker width calculation includes trailing space
List items lose indentation structure	Context stack management broken or continuation logic wrong	1. Print `ListContextStack` state changes 2. Trace context creation and finalization 3. Check continuation line indentation requirements	Fix context stack push/pop logic, correct continuation indentation requirements, ensure proper context nesting
Blockquotes miss nested levels	Blockquote prefix detection or nesting logic incorrect	1. Count `>` characters manually vs detection results 2. Check recursive blockquote parsing 3. Verify nested content processing	Fix blockquote prefix counting, ensure recursive parsing for nested quotes, handle mixed content properly
Links become literal text	Link syntax regex not matching or URL extraction failing	1. Test link patterns against failing input 2. Check bracket-parenthesis pairing 3. Verify URL extraction logic	Fix link detection regex, ensure proper bracket matching, handle edge cases like nested brackets and special characters
HTML contains unescaped characters	Character escaping missing or applied at wrong stage	1. Check when `escape_html` is called in pipeline 2. Verify all content paths include escaping 3. Look for double-escaping issues	Apply escaping consistently before HTML generation, avoid double-escaping already processed content
HTML tags are malformed or	Tag generation logic broken or	1. Check `generate_tag` output for specific elements 2. Verify self-closing vs container	Fix tag generation for self-closing elements, ensure proper

Symptom	Likely Root Cause	Diagnostic Steps	Specific Fix
unclosed	self-closing tag confusion	tag logic 3. Check tag nesting in output	opening/closing tag pairing, correct void element handling
Output HTML lacks proper indentation	Pretty printing disabled or indentation logic broken	1. Verify `pretty_print` flag is enabled 2. Check indentation level tracking 3. Test with simple nested structure	Enable pretty printing in renderer, fix indentation calculation logic, ensure consistent indentation increment
Parser crashes on empty input	Null/empty string handling missing in pipeline stages	1. Test each component with empty input 2. Check bounds checking in line processing 3. Verify null checks in parsers	Add empty input validation at pipeline entry, ensure all components handle empty/null gracefully
Infinite loops during parsing	State machine stuck or incorrect termination conditions	1. Add loop counters and limits 2. Print state transitions 3. Check termination conditions in parsing loops	Fix state transition logic, add loop guards with maximum iteration limits, ensure all parsing loops have proper exit conditions
Memory usage grows unbounded	AST nodes not properly linked or circular references created	1. Check parent-child relationships in AST 2. Look for reference cycles 3. Monitor node creation vs finalization	Fix parent pointer management, break circular references, ensure nodes are properly finalized and released
Parsing becomes extremely slow on large documents	Inefficient regex patterns or quadratic algorithm behavior	1. Profile parsing with large inputs 2. Check regex backtracking 3. Identify nested loop structures	Optimize regex patterns to avoid backtracking, replace quadratic algorithms with linear alternatives, add early termination optimizations
Inconsistent output across multiple runs	Race conditions in parser state or shared mutable state	1. Check for shared state between parser instances 2. Look for static/global variables 3. Test with concurrent parsing	Eliminate shared mutable state, ensure each parser instance is independent, make parsing functions pure

> **Critical Insight**: Most parsing bugs are actually data structure bugs in disguise. When emphasis doesn't work, it's usually because the delimiter stack isn't managed correctly. When lists break, it's because the context stack or indentation tracking has bugs. Always examine the underlying data structures first.

Advanced Diagnostic Techniques

Beyond basic symptom diagnosis, effective markdown parser debugging requires specialized techniques tailored to text processing and document transformation challenges.

AST Structure Inspection

The most powerful debugging tool for markdown parsing is systematic AST inspection. The `debug_print_ast` function should be your first resort when output doesn't match expectations. This function performs depth-first traversal of the parsed tree and displays the complete structure with proper indentation.

When using AST inspection, look for these common structural problems: missing parent-child relationships indicating incomplete tree construction, incorrect node types suggesting classification errors during parsing, unexpected nesting levels

pointing to state machine bugs, and missing or extra child nodes revealing content processing issues.

The debugging workflow involves first printing the AST after block parsing to verify block-level structure is correct, then printing again after inline parsing to check inline element nesting, and finally examining the tree structure before HTML generation to ensure all content is properly represented.

Pipeline Stage Isolation

Effective debugging requires isolating which stage of the parsing pipeline introduces errors. This systematic approach involves testing each component independently with controlled inputs.

Start by feeding known-good input to the preprocessor and verifying line splitting, normalization, and metadata extraction. Then pass preprocessor output to the block parser and examine the resulting block-level AST structure. Next, take known-good block nodes and test inline parsing in isolation. Finally, provide known-good AST structures to the HTML generator and verify output correctness.

This isolation technique quickly narrows the problem scope and prevents you from debugging the wrong component. Most developers waste time examining the HTML generator when the real bug is in block parsing or preprocessor normalization.

State Machine Debugging

Block and list parsing both use state machines that can be challenging to debug when they malfunction. The key is systematic state transition logging combined with input correlation.

For block parsing state machine issues, log every state transition with the triggering input line, current block type, and any relevant context. Create a state transition trace that shows the complete sequence of states for failing input. Look for incorrect transitions (moving to wrong state), missing transitions (staying in state when should change), or impossible states (reaching states that shouldn't be reachable with given input).

List parsing state machine debugging requires additional attention to the context stack. Log context creation, modification, and destruction events. Track indentation calculations and marker compatibility checks. The most common bugs involve incorrect context nesting or failure to properly close contexts when list structures end.

Regular Expression Testing and Optimization

Many parsing bugs stem from incorrect regular expressions that work for simple cases but fail on edge cases or complex input. Systematic regex debugging involves creating comprehensive test cases and analyzing pattern behavior.

Test each regex pattern in isolation with boundary cases: empty strings, maximum length inputs, inputs with special characters, inputs with mixed whitespace types, and inputs with unusual but valid markdown syntax. Use regex debugging tools to visualize pattern matching and identify backtracking issues that cause performance problems.

Common regex pitfalls include greedy quantifiers causing over-matching, insufficient escaping of special characters, patterns that don't account for all valid whitespace types, and patterns that fail on Unicode content or non-ASCII characters.

Domain-Specific Debugging Techniques

Text parsing and document transformation introduce unique debugging challenges that require specialized approaches beyond general software debugging techniques.

Whitespace and Character Encoding Issues

Invisible characters are a frequent source of parsing bugs that are extremely difficult to diagnose without proper techniques. Develop the habit of inspecting raw character codes rather than relying on visual inspection of text content.

Create debugging utilities that display whitespace visually by replacing spaces with `·`, tabs with `→`, and line endings with `↲`. This makes invisible formatting immediately apparent. Use hexadecimal dumps to inspect raw byte sequences when dealing with encoding issues or unusual whitespace characters.

Pay special attention to mixed line endings (mixing `\n` and `\r\n` in the same document), trailing whitespace that affects block boundary detection, tabs versus spaces in indentation calculations, and Unicode whitespace characters that don't match standard space patterns.

The `normalize_edge_case_whitespace` function should be instrumented with detailed logging showing before and after content along with explanations of what normalization was applied and why.

Context Dependency Debugging

Markdown parsing is highly context-dependent, meaning the same character sequence can have different meanings depending on surrounding content. Debugging context-dependent parsing requires understanding these dependencies and systematically testing context variations.

For emphasis parsing, the same asterisk character can be a literal character, an opening delimiter, or a closing delimiter depending on flanking context. Debug emphasis issues by creating minimal test cases that isolate specific flanking scenarios. Test emphasis with various surrounding character types: punctuation, whitespace, alphanumeric, and Unicode characters.

List parsing context dependencies involve indentation relationships between lines, marker type consistency within lists, and lazy continuation rules that allow certain lines to be included in list items without full indentation. Debug list issues by creating test cases that systematically vary indentation levels and marker combinations.

Blockquote context dependencies involve prefix detection and nested blockquote handling. Debug blockquote issues by testing various combinations of `>` characters, spaces, and content types.

Delimiter Balance and Nesting Debugging

Inline formatting relies on proper delimiter balancing that can become complex with nested emphasis, code spans that prevent emphasis processing, and multiple competing delimiter types in the same text span.

Use delimiter stack visualization to understand how delimiters are being processed. Create debugging output that shows the stack state after each character is processed, including delimiter position, type, and open/close capabilities. This visualization quickly reveals delimiter matching errors and precedence issues.

Test delimiter edge cases systematically: unmatched openers (emphasis that never closes), unmatched closers (closing emphasis without opener), overlapping delimiters of different types, delimiters inside code spans that should be treated as literal text, and complex nesting scenarios with multiple emphasis levels.

The `detect_emphasis_precedence_conflict` function should be used extensively during debugging to identify potential conflicts before they cause rendering issues.

Tree Structure and Parent-Child Relationship Debugging

AST construction bugs often manifest as incorrect parent-child relationships, missing links between nodes, or malformed tree structures that cause HTML generation to fail or produce incorrect nesting.

Implement comprehensive tree validation that checks for common structural problems: nodes with missing parent pointers, parents that don't include children in their child lists, circular references that could cause infinite loops during traversal, and orphaned nodes that exist in memory but aren't reachable from the document root.

Use tree visualization tools that display the complete AST structure with clear parent-child relationships. This visualization should include node types, content summaries, and relationship indicators that make structural problems immediately apparent.

The `validate_pipeline_state` function should be called between each major parsing phase to catch structural problems early before they propagate to later stages.

Performance Debugging for Large Documents

Markdown parsers can exhibit performance problems that only appear with large documents or complex nesting structures. These performance issues require specialized debugging approaches focused on algorithmic complexity and resource usage.

Profile parsing performance with documents of varying sizes to identify quadratic or exponential behavior. Common performance pitfalls include nested loops in list processing, backtracking regex patterns, repeated AST traversals, and inefficient string manipulation that creates many temporary objects.

Use algorithmic analysis to verify that parsing algorithms are linear in input size. Implement monitoring for excessive memory allocation, particularly in delimiter processing and AST construction. Monitor parsing time per input size to identify performance regressions.

Create performance test cases with pathological inputs that stress-test worst-case behavior: deeply nested lists, very long lines with many delimiters, documents with thousands of emphasis markers, and complex mixed formatting that exercises all parser components simultaneously.

Implementation Guidance

The debugging infrastructure for a markdown parser requires specialized tools and techniques tailored to text processing and document transformation challenges.

Technology Recommendations

Component	Simple Option	Advanced Option
AST Debugging	Print statements with manual formatting	Rich tree visualization library (anytree for Python)
Regex Testing	Online regex testers and manual verification	Integrated regex profiler with backtrack analysis
State Logging	File-based logging with manual analysis	Structured logging with real-time analysis dashboard
Performance Profiling	Manual timing with print statements	Professional profiler (cProfile for Python, pprof for Go)
Memory Analysis	Basic memory monitoring	Memory profiler with object lifecycle tracking

Debugging Infrastructure Setup

The debugging infrastructure should be built into the parser from the beginning rather than added after problems appear. This proactive approach makes debugging much more efficient when issues inevitably arise.

```
markdown_renderer/
  debug/
    __init__.py
    ast_inspector.py        ← AST visualization and validation
    pipeline_tracer.py      ← Stage-by-stage execution tracking
    regex_tester.py         ← Pattern testing and validation
    performance_monitor.py  ← Timing and memory analysis
    test_case_generator.py  ← Systematic edge case creation
  tests/
    debug/
      test_debugging_tools.py  ← Verify debugging infrastructure
      edge_cases/              ← Systematic edge case test files
        malformed_emphasis.md
        broken_list_nesting.md
        invalid_link_syntax.md
  src/
    parser/
      block_parser.py      ← Include debug hooks in implementation
      inline_parser.py     ← Include debug hooks in implementation
```

AST Inspector Implementation

PYTHON

```python
"""
AST debugging and validation utilities for markdown parser development.
Provides comprehensive tree inspection, validation, and visualization tools.
"""

import sys
from typing import Dict, List, Optional, Set, Tuple
from dataclasses import dataclass
from enum import Enum

@dataclass
class ValidationError:
    """Represents an AST structural validation error."""
    node_id: str
    error_type: str
    description: str
    severity: str
    suggested_fix: str

class ASTValidator:
    """Validates AST structure for common construction errors."""

    def __init__(self):
        self.errors = []
        self.visited_nodes = set()
        self.node_ids = {}

    def validate_tree(self, root: 'ASTNode') -> List[ValidationError]:
        """
        Perform comprehensive AST validation checking for structural problems.
        Returns list of validation errors found during inspection.
```

```
        """
        # TODO 1: Clear validation state for new validation run
        # TODO 2: Assign unique IDs to all nodes for error reporting
        # TODO 3: Check for circular references using depth-first traversal
        # TODO 4: Validate parent-child relationship consistency
        # TODO 5: Check for orphaned nodes not reachable from root
        # TODO 6: Verify node type constraints and content validation
        # TODO 7: Check for missing required attributes in block/inline nodes
        # TODO 8: Validate proper nesting according to CommonMark rules
        pass

    def check_parent_child_consistency(self, node: 'ASTNode') -> None:
        """Verify bidirectional parent-child relationships are correct."""
        # TODO 1: For each child, verify child.parent points to this node
        # TODO 2: Verify all children in child list are unique
        # TODO 3: Check that parent's child list includes this node
        # TODO 4: Recursively validate all descendant nodes
        pass

def debug_print_ast(node: 'ASTNode', indent: int = 0) -> None:
    """
    Print comprehensive AST structure with detailed node information.
    Shows node types, content, attributes, and relationships clearly.
    """
    # TODO 1: Print node basic information (type, line number, content summary)
    # TODO 2: Print node-specific attributes (block_type, inline_type, etc.)
    # TODO 3: Print relationship information (parent type, child count)
    # TODO 4: Print content preview (first 50 chars of text content)
    # TODO 5: Recursively print all children with increased indentation
    # TODO 6: Add visual indicators for tree structure (├── └──)
    pass
```

```
def find_parsing_divergence(expected_ast: 'ASTNode', actual_ast: 'ASTNode') -> List[str]:
    """
    Compare expected vs actual AST structures and identify differences.
    Useful for test failure analysis and regression debugging.
    """
    # TODO 1: Compare node types at each level of tree structure
    # TODO 2: Compare content and attributes for matching nodes
    # TODO 3: Identify missing, extra, or differently typed children
    # TODO 4: Generate detailed difference report with specific locations
    # TODO 5: Suggest likely causes for each identified difference
    pass
```

Pipeline State Tracer

```python
"""
Pipeline execution tracing for step-by-step parsing analysis.
Tracks data flow through all parsing stages with detailed state capture.
"""

from typing import Any, Dict, List, Optional
from dataclasses import dataclass, field
from datetime import datetime

@dataclass
class PipelineEvent:
    """Records a single event in the parsing pipeline execution."""
    timestamp: datetime
    stage: str
    event_type: str
    input_summary: str
    output_summary: str
    state_changes: Dict[str, Any]
    performance_metrics: Dict[str, float]
    debug_context: Dict[str, Any] = field(default_factory=dict)

class PipelineTracer:
    """Traces complete parsing pipeline execution for debugging analysis."""

    def __init__(self, enable_detailed_tracing: bool = True):
        self.events = []
        self.detailed_tracing = enable_detailed_tracing
        self.stage_timers = {}
        self.current_context = {}

    def trace_stage_entry(self, stage_name: str, input_data: Any, initial_state: Dict[str, Any]) -> None:
```

```
    """Record entry into a parsing stage with input and initial state."""
    # TODO 1: Record stage entry timestamp and input characteristics
    # TODO 2: Capture initial state snapshot for comparison
    # TODO 3: Start performance timer for this stage
    # TODO 4: Create stage context for subsequent event tracking
    pass

def trace_state_change(self, stage_name: str, change_description: str,
                       old_state: Any, new_state: Any, triggering_input: str) -> None:
    """Record significant state changes during parsing."""
    # TODO 1: Compare old and new state to identify specific changes
    # TODO 2: Record triggering input that caused the state change
    # TODO 3: Calculate and record timing since last state change
    # TODO 4: Add contextual information about why change occurred
    pass

def generate_execution_report(self) -> str:
    """Generate comprehensive report of pipeline execution for analysis."""
    # TODO 1: Summarize overall pipeline execution timing and stages
    # TODO 2: Identify performance bottlenecks and slow operations
    # TODO 3: Highlight unusual state transitions or unexpected events
    # TODO 4: Provide recommendations for performance or correctness improvements
    pass
```

Regex Pattern Tester

```python
"""
Regular expression testing and validation utilities for markdown parsing patterns.
Provides comprehensive pattern testing with edge case generation and performance analysis.
"""

import re
from typing import List, Tuple, Dict, Optional
from dataclasses import dataclass

@dataclass
class RegexTestCase:
    """Test case for regex pattern validation."""
    name: str
    input_text: str
    should_match: bool
    expected_groups: Optional[List[str]] = None
    expected_span: Optional[Tuple[int, int]] = None
    edge_case_category: str = "normal"

class MarkdownRegexTester:
    """Comprehensive testing framework for markdown parsing regex patterns."""

    def __init__(self):
        self.test_cases = []
        self.pattern_performance = {}

    def test_pattern_comprehensive(self, pattern: re.Pattern, pattern_name: str) -> Dict[str, Any]:
        """
        Test regex pattern against comprehensive set of edge cases.
        Returns detailed analysis of pattern behavior and potential issues.
        """
```

```
    # TODO 1: Generate systematic edge case test inputs
    # TODO 2: Test pattern against each case and record results
    # TODO 3: Identify false positives and false negatives
    # TODO 4: Analyze pattern performance with various input sizes
    # TODO 5: Check for catastrophic backtracking scenarios
    # TODO 6: Generate detailed test report with recommendations
    pass


def create_emphasis_edge_cases(self) -> List[RegexTestCase]:
    """Create comprehensive edge cases for emphasis delimiter testing."""
    # TODO 1: Create intraword underscore test cases
    # TODO 2: Create nested emphasis combinations
    # TODO 3: Create delimiter precedence conflict cases
    # TODO 4: Create flanking detection boundary cases
    # TODO 5: Create Unicode and special character cases
    pass


def create_list_marker_edge_cases(self) -> List[RegexTestCase]:
    """Create comprehensive edge cases for list marker detection."""
    # TODO 1: Create various whitespace combinations after markers
    # TODO 2: Create invalid marker combinations that should not match
    # TODO 3: Create nested indentation and continuation scenarios
    # TODO 4: Create mixed ordered/unordered marker test cases
    pass
```

Performance and Memory Profiler

```python
"""
Performance monitoring and memory analysis for markdown parser optimization.
Tracks parsing performance across document sizes and complexity levels.
"""

import time
import tracemalloc
from typing import Dict, List, Any, Optional, Callable
from dataclasses import dataclass, field
from collections import defaultdict

@dataclass
class PerformanceMetrics:
    """Performance measurements for parsing operations."""
    operation_name: str
    execution_time: float
    memory_peak: int
    memory_current: int
    input_size: int
    complexity_factors: Dict[str, int] = field(default_factory=dict)

class MarkdownParserProfiler:
    """Comprehensive performance profiling for markdown parser components."""

    def __init__(self, enable_memory_tracking: bool = True):
        self.metrics = []
        self.memory_tracking = enable_memory_tracking
        self.operation_counters = defaultdict(int)
        self.timing_stack = []

    def profile_parsing_operation(self, operation_name: str, operation_func: Callable,
```

```
                            input_data: Any, complexity_factors: Dict[str, int] = None) -> Any:
    """
    Profile a parsing operation with comprehensive performance measurement.
    Returns operation result along with detailed performance metrics.
    """
    # TODO 1: Start memory tracking if enabled
    # TODO 2: Record operation start time and input characteristics
    # TODO 3: Execute operation while monitoring resource usage
    # TODO 4: Record final memory usage and execution time
    # TODO 5: Calculate performance characteristics and complexity metrics
    # TODO 6: Store metrics for later analysis and reporting
    pass

def generate_performance_report(self, min_operation_time: float = 0.001) -> str:
    """Generate comprehensive performance analysis report."""
    # TODO 1: Analyze performance trends across input sizes
    # TODO 2: Identify operations with quadratic or worse complexity
    # TODO 3: Highlight memory usage patterns and potential leaks
    # TODO 4: Compare performance across different input characteristics
    # TODO 5: Provide optimization recommendations based on profiling data
    pass

def create_performance_test_suite(self) -> List[Tuple[str, str, Dict[str, int]]]:
    """Create systematic performance test cases with varying complexity."""
    # TODO 1: Create documents with varying sizes (small to very large)
    # TODO 2: Create documents with different nesting complexity levels
    # TODO 3: Create documents with varying density of formatting elements
    # TODO 4: Create pathological cases that stress-test worst-case behavior
    pass
```

Milestone Debugging Checkpoints

After completing each milestone, verify parsing correctness using these systematic checkpoints:

Milestone 1 (Block Elements) Debugging Checkpoint:

1. Run `python -m debug.ast_inspector tests/milestone1_blocks.md` and verify AST structure matches expected block hierarchy
2. Test edge cases: headings with trailing spaces, code blocks with unusual fencing, blockquotes with inconsistent prefixes
3. Check state machine transitions using `PipelineTracer` - verify clean transitions between block types
4. Expected behavior: All block-level structures parse correctly, no content is lost or misclassified

Milestone 2 (Inline Elements) Debugging Checkpoint:

1. Run delimiter stack visualization on complex emphasis examples to verify proper matching
2. Test flanking detection with edge cases using `MarkdownRegexTester`
3. Verify inline parsing doesn't interfere with block structure using AST comparison
4. Expected behavior: Emphasis, links, and inline code render correctly without breaking block structure

Milestone 3 (Lists) Debugging Checkpoint:

1. Use `debug_print_ast` to verify nested list structures are properly represented
2. Test indentation edge cases with mixed tabs/spaces and various nesting levels
3. Verify context stack management using pipeline tracer during complex list parsing
4. Expected behavior: All list nesting levels are correct, no items are lost or misplaced

Milestone 4 (HTML Generation) Debugging Checkpoint:

1. Validate generated HTML using W3C validator or similar tool
2. Check HTML entity escaping completeness using systematic character testing
3. Verify pretty printing produces properly indented, human-readable output
4. Expected behavior: Valid HTML5 output with proper escaping and formatting

Future Extensions

> **Milestone(s):** Builds upon all milestones - Milestone 1: Block Elements, Milestone 2: Inline Elements, Milestone 3: Lists, Milestone 4: HTML Generation

The Evolution Framework Mental Model

Think of extending the markdown renderer like upgrading a modular kitchen. The core appliances (block parser, inline parser, HTML generator) are already in place and working perfectly. Now you want to add specialty equipment - perhaps a pasta machine for tables, a bread maker for math expressions, or modular attachments that let third-party manufacturers create custom tools. The key insight is that these extensions should plug into the existing infrastructure without requiring you to rewire the entire kitchen. A well-designed extension system feels like adding new capabilities that were always meant to be there, rather than awkward bolt-ons that compromise the original design.

The architectural challenge with markdown extensions lies in maintaining the elegant simplicity of the core parsing pipeline while accommodating the complexity that advanced features inevitably introduce. Each extension potentially touches multiple components - new syntax requires lexer changes, new block types need parser modifications, new output formats demand

renderer updates. The goal is to create extension points that feel natural and maintain the clean separation of concerns established in the base system.

CommonMark Extensions

The CommonMark specification intentionally focuses on a stable, well-defined core that captures the essence of markdown while leaving room for extensions. These extensions represent commonly requested features that build logically on the existing parsing infrastructure. Understanding how to add these extensions reveals the flexibility and robustness of the two-phase parsing architecture.

Table Extension Architecture

Tables represent one of the most requested markdown extensions, transforming the simple grid-like syntax into structured HTML table elements. The table extension demonstrates how complex block-level structures can be parsed using the existing block parser infrastructure with minimal modifications to the core architecture.

The table parsing process follows a recognition pattern similar to existing block elements. The block parser identifies potential table structures by detecting pipe characters (`|`) that suggest columnar organization. However, unlike simple blocks like paragraphs or headings, tables require sophisticated parsing of internal structure to extract headers, alignment specifications, and cell content.

Consider how a table integrates into the existing parsing pipeline. During block parsing, the system encounters lines that contain pipe characters in positions that suggest table structure. The table parser examines consecutive lines to determine if they form a valid table structure - a header row, a delimiter row specifying column alignments, and one or more data rows. This multi-line analysis fits naturally into the block parser's lookahead capabilities.

Table Structure Recognition Algorithm:

1. **Initial Detection**: During block parsing, when encountering a line containing pipe characters, examine if this could be a table header row
2. **Delimiter Row Validation**: Look ahead to the next line to check for a valid delimiter row with alignment specifications (`:---` , `:---:` , `---:`)
3. **Structure Confirmation**: Verify that header and delimiter rows have compatible column counts and valid separator patterns
4. **Row Collection**: Continue consuming lines that match the established table structure until encountering a line that breaks the pattern
5. **Cell Content Extraction**: Parse individual cell content, trimming whitespace and preparing for inline element processing
6. **Alignment Processing**: Extract column alignment information from the delimiter row and store as table metadata
7. **AST Node Creation**: Create a table `BlockNode` with child nodes representing rows and cells, preserving alignment specifications

The table extension introduces new node types that integrate seamlessly with the existing AST structure:

Node Type	Parent Node	Child Nodes	Special Attributes
`TABLE`	Document or container block	`TABLE_HEADER` , `TABLE_BODY`	column_count, alignment_specs
`TABLE_HEADER`	`TABLE`	`TABLE_ROW`	none
`TABLE_BODY`	`TABLE`	`TABLE_ROW`	none
`TABLE_ROW`	`TABLE_HEADER` or `TABLE_BODY`	`TABLE_CELL`	row_type
`TABLE_CELL`	`TABLE_ROW`	Inline elements	alignment, is_header

The inline parsing phase processes table cell content normally, applying emphasis, links, and other inline formatting within cell boundaries. This demonstrates the power of the two-phase parsing architecture - table structure is resolved during block parsing, while cell formatting is handled during inline parsing without requiring specialized logic.

> **Design Insight**: Tables showcase how complex extensions can leverage the existing parsing infrastructure. The block parser handles structure recognition and AST construction, while the inline parser processes cell content using existing mechanisms. No modifications to the core inline parsing logic are required.

Math Expression Extension

Mathematical expressions represent a different category of extension that introduces domain-specific syntax requiring specialized rendering. Math extensions typically support both inline math (delimited by single dollar signs) and display math (delimited by double dollar signs or fenced blocks). This extension demonstrates how to handle content that requires pass-through processing rather than standard markdown parsing.

The math extension introduces the concept of **verbatim content blocks** - regions where normal markdown parsing is suspended in favor of preserving exact content for specialized processors. This is similar to how code blocks preserve content literally, but math blocks may require additional processing by mathematical typesetting systems.

Math Block Recognition Process:

1. **Inline Math Detection**: During inline parsing, detect single dollar sign delimiters that enclose mathematical expressions
2. **Display Math Block Detection**: During block parsing, detect double dollar signs or fenced math blocks (```` ```math `or` $$` ````)
3. **Content Preservation**: Extract math content without applying normal markdown parsing rules - preserve all characters literally
4. **Metadata Extraction**: Identify the math notation type (LaTeX, MathML, etc.) from fences or context
5. **AST Integration**: Create specialized math nodes that carry the unparsed mathematical content and notation metadata
6. **Renderer Integration**: Pass math content to specialized renderers (MathJax, KaTeX) while handling the HTML integration

Math extensions typically require coordination with client-side rendering libraries or server-side mathematical typesetting systems. The markdown renderer's responsibility is to preserve the mathematical content accurately and provide the necessary HTML structure for math rendering libraries to process.

Extension Feature	Block Level	Inline Level	Processing Phase	Special Requirements
Table Structure	Yes	Cell content only	Block then inline	Column alignment, row grouping
Math Expressions	Display math	Inline math	Content preservation	External renderer integration
Footnotes	Reference definitions	Reference markers	Cross-document linking	Link resolution, numbering
Task Lists	Special list items	Checkbox indicators	List parsing extension	Interactive element generation
Definition Lists	Term/definition pairs	Term highlighting	Block parsing extension	Semantic HTML generation

Task List Extension

Task lists extend the existing list parsing infrastructure to support interactive checkbox elements. This extension demonstrates how existing parsing components can be enhanced without disrupting their core functionality. Task lists use a special syntax within list items (`- [ ]` for unchecked, `- [x]` for checked) that the list parser can recognize and process.

The task list extension modifies list item recognition to detect checkbox patterns at the beginning of list item content. When such patterns are found, the list parser extracts the checkbox state and marks the list item with task list attributes. During HTML

generation, these attributes trigger the creation of interactive checkbox input elements.

> **Design Principle**: Extensions should enhance existing functionality rather than replace it. Task lists build on the standard list parsing infrastructure, adding recognition patterns and output modifications while preserving all existing list behaviors.

Architecture Decision: Extension Integration Strategy

> **Decision: Compositional Extension Architecture**
>
> - **Context**: Extensions need to add functionality without modifying core parsing logic or breaking backward compatibility
> - **Options Considered**:
>   1. **Inheritance-based extensions**: Subclass core parsers and override methods
>   2. **Plugin system**: Load extensions dynamically with registration hooks
>   3. **Compositional extensions**: Extend parsers through composition and configuration
> - **Decision**: Implement compositional extensions with registration-based integration
> - **Rationale**: Composition preserves the stability of core parsing logic while allowing extensions to add new parsing rules and output formats. Registration-based integration provides clear extension points without requiring dynamic loading complexity
> - **Consequences**: Extensions integrate cleanly with minimal core modifications, but advanced extensions may require multiple registration points across different parsing phases

Plugin and Extension Architecture

The plugin architecture provides a systematic framework for extending the markdown renderer beyond the built-in CommonMark extensions. This architecture must balance flexibility with simplicity, allowing powerful customizations without compromising the performance or reliability of the core parsing pipeline.

The Extension Point Mental Model

Think of extension points like electrical outlets in a building. The building's wiring (core parsing pipeline) provides standardized connection points where different appliances (extensions) can plug in safely. Each outlet type (extension point) provides specific electrical characteristics (data formats, calling conventions, lifecycle management) that appliances must respect. Well-designed extension points mean that new appliances can be added without rewiring the building, and multiple appliances can coexist without interference.

The markdown renderer provides several natural extension points corresponding to the major phases of document processing. Each extension point offers different capabilities and operates on different data representations, allowing extensions to target the most appropriate level of abstraction for their functionality.

Extension Point Architecture

The plugin system defines multiple extension points that correspond to different phases of the parsing pipeline. Each extension point operates on specific data structures and provides well-defined hooks for custom functionality.

Extension Point	Input Data	Output Data	Timing	Use Cases
Preprocessor Plugins	Raw markdown text	Modified text	Before any parsing	Text normalization, macro expansion
Block Parser Plugins	`LineInfo` sequences	Additional `BlockNode` types	During block parsing	Custom block syntax
Inline Parser Plugins	Text spans within blocks	Additional `InlineNode` types	During inline parsing	Custom inline formatting
Renderer Plugins	`ASTNode` instances	HTML fragments	During HTML generation	Custom output formats
Post-processor Plugins	Complete HTML	Modified HTML	After rendering	Document-wide transformations

The `PluginManager` coordinates plugin registration and execution, ensuring that plugins are invoked at appropriate points in the parsing pipeline. The manager maintains plugin registries for each extension point and provides the interface for plugins to declare their capabilities and requirements.

Plugin Interface Design

Each plugin type implements a specific interface that defines its interaction with the parsing pipeline. These interfaces are designed to be minimal yet powerful, allowing plugins to integrate seamlessly while maintaining clear separation of concerns.

Renderer Plugin Interface:

Method	Parameters	Returns	Purpose
`get_supported_types()`	None	`List[NodeType]`	Declare which AST node types this plugin handles
`render_node(node, context)`	`ASTNode`, `RenderContext`	`str`	Convert AST node to output format
`get_priority()`	None	`int`	Determine plugin precedence for overlapping capabilities
`initialize(config)`	`Dict[str, Any]`	`bool`	Perform plugin initialization with configuration
`finalize()`	None	`None`	Clean up plugin resources

The `RenderContext` provides plugins with access to the rendering environment, including the parent renderer, configuration options, and utility functions for common tasks like HTML escaping and indentation management.

Block Parser Plugin Interface:

Method	Parameters	Returns	Purpose
`can_handle_block(line)`	`LineInfo`	`bool`	Determine if plugin can parse a block starting with this line
`parse_block(lines, start_index)`	`List[LineInfo]`, `int`	`Tuple[BlockNode, int]`	Parse block and return consumed line count
`get_block_precedence()`	None	`int`	Priority for block recognition conflicts

Plugin Configuration and Lifecycle

The plugin system supports both built-in extensions that ship with the renderer and external plugins loaded dynamically. Plugin configuration allows users to enable, disable, and configure individual plugins without modifying core code.

Plugin Configuration Structure:

```
Plugin Configuration:
  - plugin_name: str (unique identifier)
  - enabled: bool (activation status)
  - priority: int (execution order)
  - config: Dict[str, Any] (plugin-specific settings)
  - dependencies: List[str] (required plugins)
  - conflicts: List[str] (incompatible plugins)
```

The `PluginManager` handles plugin lifecycle management, ensuring that plugins are loaded in dependency order, initialized with appropriate configuration, and cleaned up properly when the renderer shuts down.

Plugin Lifecycle States:

State	Description	Allowed Transitions	Cleanup Required
`UNLOADED`	Plugin not yet loaded	→ `LOADING`	No
`LOADING`	Plugin being initialized	→ `LOADED`, `ERROR`	Partial
`LOADED`	Plugin ready for use	→ `ACTIVE`, `UNLOADING`	No
`ACTIVE`	Plugin currently processing	→ `LOADED`	No
`ERROR`	Plugin failed to load/initialize	→ `UNLOADING`	Yes
`UNLOADING`	Plugin being shut down	→ `UNLOADED`	Yes

Custom Renderer Development

Custom renderers demonstrate the most common and powerful use of the plugin architecture. Rather than generating HTML, custom renderers can produce LaTeX for academic papers, plain text for accessibility, or structured data formats like JSON or XML.

A LaTeX renderer plugin illustrates how specialized output formats leverage the existing AST structure while producing completely different markup. The LaTeX renderer traverses the same AST generated by the standard parsing pipeline but emits LaTeX commands instead of HTML tags.

LaTeX Renderer Example Mapping:

AST Node Type	HTML Output	LaTeX Output	Special Handling
`HEADING` (level 1)	`<h1>title</h1>`	`\section{title}`	Convert heading levels to LaTeX sectioning
`STRONG`	`<strong>text</strong>`	`\textbf{text}`	Handle nested emphasis
`EMPHASIS`	`<em>text</em>`	`\textit{text}`	Handle nested emphasis
`CODE_BLOCK`	`<pre><code>content</code></pre>`	`\begin{verbatim}content\end{verbatim}`	Preserve exact formatting
`LINK`	`<a href="url">text</a>`	`\href{url}{text}`	URL validation for LaTeX

Custom renderers must handle the tree traversal logic and maintain proper nesting of their output format. The plugin interface provides helper methods for common tasks, but complex renderers may need sophisticated state management to produce well-formed output.

Extension Development Best Practices

Effective plugin development requires understanding both the markdown renderer's architecture and the specific requirements of the extension's domain. Successful plugins follow several key principles that ensure reliable integration and good performance.

Plugin Development Principles:

1. **Minimal Surface Area**: Plugins should interact with the core system through well-defined interfaces, avoiding dependencies on internal implementation details
2. **Error Isolation**: Plugin errors should not crash the entire parsing pipeline - implement proper error handling and graceful degradation
3. **Performance Awareness**: Plugins operate within the parsing pipeline and can impact overall performance - optimize for common cases and avoid expensive operations in hot paths
4. **Configuration Validation**: Validate plugin configuration early and provide clear error messages for misconfigurations
5. **Documentation Standards**: Provide clear documentation of supported syntax, configuration options, and interaction with other plugins

> **Architecture Insight**: The plugin architecture's power comes from operating on the AST representation rather than raw text. This means plugins can focus on their specific functionality without reimplementing text parsing logic, and multiple plugins can collaborate by operating on the same standardized data structures.

Extension Ecosystem Considerations

As the plugin ecosystem grows, certain patterns emerge for managing plugin interactions and maintaining system stability. The plugin architecture must accommodate both simple single-purpose plugins and complex multi-feature extensions.

Plugin Interaction Patterns:

Pattern	Description	Benefits	Challenges
Independent Plugins	Plugins operate on disjoint node types	No conflicts, simple testing	Limited collaboration possibilities
Layered Processing	Plugins process in defined order with dependencies	Powerful compositions, clear data flow	Complex dependency management
Collaborative Plugins	Plugins share state or coordinate through APIs	Rich feature interactions	Increased coupling, harder debugging
Override Plugins	Later plugins can replace earlier plugin output	Flexible customization	Potential conflicts, unclear precedence

The plugin manager implements sophisticated conflict detection and resolution to handle cases where multiple plugins claim to handle the same node types or syntax patterns. Priority systems and explicit conflict declarations help users configure complex plugin combinations successfully.

Implementation Guidance

Technology Recommendations

Component	Simple Implementation	Advanced Implementation
Extension Registry	Dictionary-based lookup with static registration	Dynamic loading with dependency resolution
Plugin Interface	Duck-typed protocols	Abstract base classes with type checking
Configuration Management	JSON/YAML files with manual validation	Schema validation with automatic documentation
Plugin Discovery	Explicit registration in main module	Automatic discovery via entry points
Error Handling	Try-catch around plugin calls	Circuit breaker pattern with fallback strategies

File Structure for Extensions

```
project-root/
  core/
    parser.py           ← Core parsing logic
    ast_nodes.py        ← AST node definitions
    html_renderer.py    ← Base HTML renderer

  extensions/
    __init__.py         ← Extension registry
    tables.py           ← Table extension
    math.py             ← Math expression extension
    task_lists.py       ← Task list extension

  plugins/
    __init__.py         ← Plugin manager and interfaces
    base_plugin.py      ← Abstract plugin base classes
    renderer_plugins.py ← Renderer plugin interfaces
    parser_plugins.py   ← Parser plugin interfaces

  renderers/
    latex_renderer.py   ← LaTeX output renderer
    json_renderer.py    ← JSON AST serializer
    plain_text.py       ← Accessibility text renderer
```

Plugin Manager Infrastructure

```python
from abc import ABC, abstractmethod

from typing import Dict, List, Any, Optional, Protocol

from enum import Enum

import logging


class PluginState(Enum):

    UNLOADED = "unloaded"

    LOADING = "loading"

    LOADED = "loaded"

    ACTIVE = "active"

    ERROR = "error"

    UNLOADING = "unloading"


class RendererPlugin(ABC):

    """

    Abstract base class for custom renderer plugins.

    Renderers transform AST nodes into output formats.

    """



    @abstractmethod

    def get_supported_types(self) -> List[str]:

        """Return list of NodeType enum values this plugin handles."""

        # TODO: Return NodeType values this renderer supports

        pass



    @abstractmethod

    def render_node(self, node: 'ASTNode', context: 'RenderContext') -> str:

        """

        Render a single AST node to output format.



        Args:
```

```
            node: AST node to render
            context: Rendering context with utilities and state

        Returns:
            String representation in target format
        """
        # TODO: Implement node-specific rendering logic
        # TODO: Handle child node rendering via context.render_children()
        # TODO: Apply any node-specific formatting or escaping
        pass

    def get_priority(self) -> int:
        """Return priority for plugin precedence (higher = more priority)."""
        return 0

    def initialize(self, config: Dict[str, Any]) -> bool:
        """Initialize plugin with configuration. Return True if successful."""
        # TODO: Validate configuration parameters
        # TODO: Set up any required resources or connections
        # TODO: Return False if initialization fails
        return True

    def finalize(self) -> None:
        """Clean up plugin resources."""
        # TODO: Close any open files, connections, or resources
        # TODO: Save any persistent state if needed
        pass

class PluginManager:
    """
    Manages plugin registration, lifecycle, and execution.
```

```
    Coordinates between core parser and extension plugins.
    """

    def __init__(self):
        self.plugins: Dict[str, RendererPlugin] = {}
        self.node_renderers: Dict[str, List[RendererPlugin]] = {}
        self.plugin_states: Dict[str, PluginState] = {}
        self.plugin_configs: Dict[str, Dict[str, Any]] = {}
        self.logger = logging.getLogger(__name__)

    def register_plugin(self, name: str, plugin: RendererPlugin,
                        config: Optional[Dict[str, Any]] = None) -> bool:
        """
        Register a renderer plugin with the manager.

        Args:
            name: Unique plugin identifier
            plugin: Plugin instance implementing RendererPlugin
            config: Optional configuration dictionary

        Returns:
            True if registration successful, False otherwise
        """
        # TODO: Check if plugin name already exists
        # TODO: Validate plugin implements required interface
        # TODO: Store plugin with UNLOADED state
        # TODO: Initialize plugin with provided config
        # TODO: Update node_renderers mapping for supported types
        # TODO: Handle initialization errors gracefully
        pass
```

```
    def get_renderer(self, node_type: str) -> Optional[RendererPlugin]:
        """
        Get the best renderer for a given node type.

        Args:
            node_type: NodeType enum value as string

        Returns:
            Highest priority renderer for node type, or None
        """
        # TODO: Look up renderers for node_type in node_renderers
        # TODO: Return highest priority renderer from available options
        # TODO: Handle case where no renderer is available
        pass

    def render_with_fallback(self, node: 'ASTNode',
                             context: 'RenderContext') -> str:
        """
        Render node with automatic fallback to default renderer.

        Args:
            node: AST node to render
            context: Rendering context

        Returns:
            Rendered output with fallback if plugin fails
        """
        # TODO: Get best renderer for node.node_type
        # TODO: Attempt rendering with primary renderer
        # TODO: Catch and log any plugin errors
        # TODO: Fall back to default renderer if plugin fails
```

```
        # TODO: Return rendered output or error placeholder
        pass

class RenderContext:
    """
    Provides rendering utilities and state management for plugins.
    Passed to renderer plugins during node processing.
    """

    def __init__(self, plugin_manager: PluginManager, config: Dict[str, Any]):
        self.plugin_manager = plugin_manager
        self.config = config
        self.depth = 0
        self.current_list_context: Optional['ListContext'] = None
        self.html_escaper = HtmlEscaper()

    def render_children(self, node: 'ASTNode') -> str:
        """
        Render all child nodes of the given node.

        Args:
            node: Parent node whose children should be rendered

        Returns:
            Concatenated rendering of all child nodes
        """
        # TODO: Iterate through node.children
        # TODO: Render each child using plugin_manager.render_with_fallback
        # TODO: Concatenate results and return combined output
        pass
```

```
    def escape_for_format(self, text: str, format_type: str = "html") -> str:
        """
        Escape text for target output format.

        Args:
            text: Raw text to escape
            format_type: Target format ("html", "latex", "xml", etc.)

        Returns:
            Properly escaped text for target format
        """
        # TODO: Implement format-specific escaping logic
        # TODO: Handle HTML entities, LaTeX special chars, etc.
        # TODO: Return escaped text appropriate for format
        pass
```

Table Extension Implementation

```python
import re

from typing import List, Optional, Tuple

from dataclasses import dataclass


@dataclass

class TableAlignment:

    """Represents column alignment specification from table delimiter row."""

    left: bool = False

    right: bool = False

    center: bool = False



    @classmethod

    def from_delimiter(cls, delimiter: str) -> 'TableAlignment':

        """Parse alignment from delimiter cell like ':---:', '---:', ':---'"""

        # TODO: Check for leading colon (left alignment)

        # TODO: Check for trailing colon (right alignment)

        # TODO: Determine center alignment (both colons present)

        # TODO: Return TableAlignment instance with appropriate flags

        pass


class TableExtension:

    """

    Extension for parsing GitHub Flavored Markdown tables.

    Integrates with block parser to recognize and process table syntax.

    """



    # Regex patterns for table recognition

    TABLE_ROW_PATTERN = re.compile(r'^\s*\|.*\|\s*$')

    DELIMITER_ROW_PATTERN = re.compile(r'^\s*\|?\s*:?-+:?\s*(\|\s*:?-+:?\s*)*\|?\s*$')



    def can_start_table(self, current_line: 'LineInfo',
```

```
                    next_line: Optional['LineInfo']) -> bool:
        """
        Determine if current line could start a table.

        Args:
            current_line: Potential table header line
            next_line: Potential delimiter row (must be present)

        Returns:
            True if these lines form valid table start
        """
        # TODO: Check current_line matches TABLE_ROW_PATTERN
        # TODO: Check next_line exists and matches DELIMITER_ROW_PATTERN
        # TODO: Verify column count compatibility between header and delimiter
        # TODO: Return True only if valid table structure detected
        pass

    def parse_table_block(self, lines: List['LineInfo'],
                          start_index: int) -> Tuple['BlockNode', int]:
        """
        Parse complete table from line sequence.

        Args:
            lines: Full document line sequence
            start_index: Index of table header line

        Returns:
            Tuple of (table BlockNode, number of lines consumed)
        """
        # TODO: Parse header row and extract cell content
        # TODO: Parse delimiter row and extract alignment specifications
```

```
        # TODO: Continue parsing data rows until table structure breaks
        # TODO: Create table BlockNode with appropriate child structure
        # TODO: Set table attributes (column_count, alignments)
        # TODO: Return table node and consumed line count
        pass

    def parse_table_row(self, line: 'LineInfo',
                        expected_columns: int) -> List[str]:
        """
        Parse a single table row into cell contents.

        Args:
            line: Line containing table row
            expected_columns: Expected number of columns

        Returns:
            List of cell contents (trimmed strings)
        """
        # TODO: Split line on pipe characters
        # TODO: Handle escaped pipes (\|) that should not split
        # TODO: Trim whitespace from each cell
        # TODO: Pad or truncate to match expected_columns
        # TODO: Return list of cell content strings
        pass

    def extract_alignments(self, delimiter_line: 'LineInfo') -> List[TableAlignment]:
        """
        Extract column alignment specifications from delimiter row.

        Args:
            delimiter_line: Table delimiter row like '|:---|---:|:---:|'
```

```
        Returns:
            List of TableAlignment objects for each column
        """
        # TODO: Split delimiter line on pipe characters
        # TODO: Parse each delimiter cell for colon positions
        # TODO: Create TableAlignment objects based on colon placement
        # TODO: Return list of alignments matching column order
        pass
```

Milestone Checkpoints

Extension Integration Checkpoint: After implementing the basic plugin architecture:

1. **Plugin Registration Test**: Create a simple test plugin and verify it registers correctly

```bash
python -m pytest tests/test_plugin_manager.py::test_plugin_registration
```

2. **Table Extension Test**: Test table parsing with various alignment combinations

```bash
echo "| Header 1 | Header 2 | Header 3 |
|:---------|:--------:|---------:|
| Left     | Center   | Right    |" | python -m markdown_renderer --extension tables
```

3. **Custom Renderer Test**: Implement a simple JSON renderer and verify AST serialization

```bash
echo "# Test heading\n\nSome **bold** text" | python -m markdown_renderer --renderer json
```

Plugin Development Checkpoint: Expected plugin development workflow:

1. **Interface Implementation**: Plugin implements required abstract methods without errors
2. **Registration Success**: Plugin registers with manager and appears in plugin list
3. **Node Processing**: Plugin correctly processes assigned node types
4. **Error Handling**: Plugin failures don't crash the parser pipeline
5. **Configuration Loading**: Plugin accepts and validates configuration parameters

Common Plugin Issues:

Symptom	Likely Cause	How to Diagnose	Fix
Plugin not found	Registration failed	Check plugin manager logs	Verify plugin registration call
Node not processed	Type mapping incorrect	Check get_supported_types()	Fix node type constants
Rendering errors	Missing context usage	Test render_node() directly	Use context.render_children()
Performance slow	Inefficient traversal	Profile plugin execution	Optimize hot path operations
Config errors	Validation missing	Check initialize() return	Add config parameter validation

Glossary

> **Milestone(s):** Milestone 1: Block Elements, Milestone 2: Inline Elements, Milestone 3: Lists, Milestone 4: HTML Generation

The Reference Manual Mental Model

Think of this glossary as a reference manual for a complex piece of machinery - the markdown renderer. Just as a mechanic needs to understand the precise terminology for each engine component to diagnose problems and communicate solutions, developers working with markdown parsers need a shared vocabulary of technical terms, parsing concepts, and architectural patterns. Each term in this glossary represents a specific concept with precise meaning within the context of document transformation systems.

The glossary serves multiple purposes beyond simple definition lookup. It establishes consistent terminology across the entire system, prevents confusion between similar concepts (like "block-level elements" versus "block nodes"), and provides the conceptual foundation for understanding complex interactions between parser components. When debugging a delimiter matching issue or discussing extension architecture, having precise terminology eliminates ambiguity and accelerates problem resolution.

This reference organizes terminology into logical categories that mirror the system architecture: core parsing concepts, data structures and types, algorithms and processes, architectural patterns, error handling terminology, testing and debugging vocabulary, and extensibility concepts. Each category builds upon previous ones, creating a coherent knowledge framework for the entire markdown rendering domain.

Core Parsing Concepts

These fundamental concepts form the theoretical foundation for understanding how markdown parsers transform textual input into structured output. Each concept represents a key insight about the nature of document transformation that influences architectural decisions throughout the system.

Abstract Syntax Tree (AST): A hierarchical tree data structure that represents the parsed structure of a markdown document, where each node corresponds to a markdown element (paragraph, heading, emphasis, link, etc.) and the tree relationships capture the nesting and containment relationships between elements. The AST serves as the intermediate representation that decouples parsing logic from HTML generation, enabling different output formats and transformation passes.

Block-level elements: Structural markdown elements that define document layout and organization, including paragraphs, headings, lists, code blocks, blockquotes, and horizontal rules. Block-level elements typically span multiple lines, can contain other block elements or inline elements, and establish the primary document structure that determines reading flow and semantic meaning.

Inline elements: Formatting markdown elements that exist within block-level elements and modify text appearance or behavior, including emphasis (bold and italic), inline code spans, links, images, and line breaks. Inline elements cannot contain block-level elements and typically span portions of a single line, though they can contain other inline elements through nesting.

Two-phase parsing: A parsing architecture that separates document processing into distinct block parsing and inline parsing phases, where block parsing first identifies and structures document-level elements, then inline parsing processes formatting within each block's content. This separation simplifies parsing logic, enables context-dependent inline processing, and allows independent optimization of each parsing phase.

CommonMark: The standardized specification for markdown syntax that defines precise parsing rules, handles edge cases consistently, and provides a reference implementation for validation. CommonMark eliminates ambiguities in original markdown specifications and serves as the authoritative source for parsing behavior in complex scenarios.

Context-dependent parsing: Parsing rules that change behavior based on surrounding content, document state, or position within other elements. Examples include emphasis delimiters that behave differently at word boundaries, list markers that require specific indentation contexts, and inline code that disables other formatting rules within its boundaries.

State machine: A parsing approach that models the parser as having distinct internal states (like "looking for block," "in paragraph," "in code block") with defined transitions between states triggered by specific input patterns. State machines provide predictable parsing behavior, simplify handling of complex multi-line constructs, and enable systematic testing of parser logic.

Lookahead: The technique of examining upcoming input characters or lines without consuming them from the input stream, enabling the parser to make decisions based on future context. Lookahead is essential for handling Setext headings (which require examining the following line), determining block continuation, and resolving parsing ambiguities.

Markdown Syntax Terminology

These terms describe specific markdown syntactic constructs and their parsing requirements, providing precise vocabulary for discussing parsing challenges and implementation strategies.

ATX headings: Hash-prefix style headings that use one to six `#` characters at the beginning of a line to indicate heading levels, such as `# Heading 1` or `### Heading 3`. ATX headings are self-contained on a single line and can be identified immediately without lookahead.

Setext headings: Underline-style headings that use lines of `=` characters (for level 1) or `-` characters (for level 2) underneath the heading text. Setext headings require lookahead parsing since the heading text appears first, followed by the underline marker on the next line.

Fenced code blocks: Multi-line code blocks delimited by lines containing three or more backticks (```` ``` ````) or tildes (`~~~`), optionally followed by a language identifier for syntax highlighting. Fenced code blocks preserve all internal content literally, including markdown syntax that would otherwise be parsed.

Indented code blocks: Multi-line code blocks created by indenting every line by at least four spaces or one tab character. Indented code blocks end when a line is encountered that is not indented sufficiently or when the document ends.

Lazy continuation: The CommonMark rule allowing certain block elements (like blockquotes and list items) to continue on subsequent lines even when those lines don't include the expected prefix markers, as long as the content would otherwise be part of a paragraph.

Intraword underscore: Underscore characters that appear within a word (like `snake_case_variable`) and should not trigger emphasis formatting according to CommonMark rules. Detecting intraword underscores requires examining characters before and after potential emphasis delimiters.

Parsing Algorithm Terminology

These terms describe the specific algorithms and techniques used throughout the parsing pipeline, providing precise vocabulary for discussing implementation strategies and optimization approaches.

Delimiter matching: The algorithm for finding pairs of formatting markers (like `**` for bold or `[` and `]` for links) that define the boundaries of inline formatting. Delimiter matching must handle nesting, precedence rules, and various edge cases like unmatched or overlapping delimiters.

Delimiter stack: A data structure that tracks unmatched opening delimiters during inline parsing, enabling proper nesting and precedence handling for emphasis, strong emphasis, and other delimiter-based formatting. The delimiter stack is processed using specific algorithms defined in the CommonMark specification.

Flanking detection: The algorithm for determining whether emphasis delimiters (asterisks and underscores) can open or close emphasis based on the characters immediately before and after the delimiter sequence. Flanking rules prevent emphasis in contexts like `a*b*c` where the asterisks are surrounded by alphanumeric characters.

Emphasis delimiter precedence: The rules for resolving conflicts when multiple emphasis delimiters could match the same text span, such as in `***bold and italic***` where delimiters must be matched according to specific precedence rules to produce consistent output.

Depth-first traversal: The tree traversal algorithm used during HTML generation to visit AST nodes, where each node is processed before its siblings, and all descendants are visited before moving to the next sibling. This traversal pattern ensures proper HTML nesting and content ordering.

HTML entity escaping: The process of converting special characters (like `<`, `>`, `&`, and quotes) to their corresponding HTML entities (`<`, `>`, `&`, etc.) to prevent HTML injection and ensure proper display of literal characters.

Pretty printing: The process of formatting HTML output with consistent indentation, line breaks, and spacing to make the generated HTML human-readable for debugging and development purposes.

Data Structure and Type Terminology

These terms describe the specific data structures, types, and interfaces used throughout the system, providing precise vocabulary for discussing implementation details and component interactions.

AST node types: The enumerated categories of nodes in the abstract syntax tree, including `DOCUMENT` (root node), `BLOCK` (block-level elements), `INLINE` (inline formatting elements), and `TEXT` (literal text content). Each node type determines processing behavior and available operations.

Block types: The specific categories of block-level elements, including `PARAGRAPH`, `HEADING`, `CODE_BLOCK`, `BLOCKQUOTE`, `HORIZONTAL_RULE`, `LIST`, `LIST_ITEM`, and extended types like `TABLE` components. Block types determine parsing rules, HTML output generation, and containment relationships.

Inline types: The specific categories of inline formatting elements, including `STRONG` (bold), `EMPHASIS` (italic), `CODE` (inline code), `LINK`, `IMAGE`, and `LINE_BREAK`. Inline types determine delimiter matching rules, nesting behavior, and HTML generation logic.

Parser state enumeration: The defined states for the block parsing state machine, including `LOOKING_FOR_BLOCK`, `IN_PARAGRAPH`, `IN_FENCED_CODE`, `IN_INDENTED_CODE`, and `IN_BLOCKQUOTE`. These states determine how input lines are processed and when state transitions occur.

Delimiter types: The categories of characters that can act as formatting delimiters, including `ASTERISK`, `UNDERSCORE`, `BACKTICK`, `LEFT_BRACKET`, and `EXCLAMATION_BRACKET`. Each delimiter type has specific rules for opening and closing formatting spans.

List marker types: The categories of list markers, including `UNORDERED_DASH`, `UNORDERED_ASTERISK`, `UNORDERED_PLUS` for unordered lists, and `ORDERED_PERIOD`, `ORDERED_PAREN` for ordered lists. Marker types determine list compatibility and nesting rules.

List state enumeration: The states that track whether lists should have tight or loose formatting, including `TIGHT` (no blank lines between items), `LOOSE` (blank lines create paragraph wrapping), and `UNDETERMINED` (final state not yet decided based on input).

Error Handling and Recovery Terminology

These terms describe the systematic approach to handling malformed input, parsing errors, and recovery strategies that maintain system stability while providing useful feedback to users.

Parser error recovery: The comprehensive strategy for handling invalid markdown syntax while continuing to parse subsequent content, including detection of errors, classification by severity, application of recovery strategies, and collection of diagnostic information for user feedback.

Graceful degradation: The design principle where parser failures in one component or with one piece of input do not prevent processing of other components or content, ensuring the system produces useful output even when encountering unexpected input patterns.

Recovery strategies: The defined approaches for handling specific types of parsing errors, including `COMPLETE_BLOCK` (finish current block and continue), `CONVERT_TO_PARAGRAPH` (treat malformed syntax as regular text), `PRESERVE_CONTENT` (maintain content while fixing structure), and `SKIP_MALFORMED` (ignore invalid sections entirely).

Error severity levels: The classification system for parsing errors, including `WARNING` (non-critical issues that don't prevent processing), `ERROR` (significant problems that require recovery), and `CRITICAL` (failures that prevent further processing of affected sections).

Continuation-based recovery: The recovery approach where errors in parsing one block element do not prevent successful parsing of subsequent blocks, maintaining document-level processing even when individual elements contain syntax errors.

Delimiter balancing recovery: The specialized recovery strategy for handling unmatched or incorrectly nested emphasis delimiters, including conversion to literal characters, automatic closing of unmatched openers, and precedence-based resolution of conflicting matches.

Semantic whitespace preservation: The recovery technique that preserves whitespace characters that affect document meaning and display while normalizing insignificant whitespace, ensuring consistent output regardless of input formatting variations.

Testing and Debugging Terminology

These terms describe the specialized vocabulary for systematic testing and debugging of parser components, providing precise language for discussing quality assurance approaches and problem diagnosis techniques.

State machine debugging: The systematic debugging approach for parser state transitions, including verification of state consistency, validation of transition triggers, identification of unreachable states, and detection of infinite loops or incorrect state changes.

Pipeline stage isolation: The debugging technique that tests individual parsing components in isolation from the complete pipeline, enabling precise identification of issues within specific transformation stages without interference from other components.

Delimiter balance and nesting debugging: The specialized debugging approach for inline formatting issues, focusing on delimiter stack state, matching algorithms, precedence resolution, and the interaction between different types of emphasis delimiters.

Tree structure validation: The comprehensive checking of AST parent-child relationships, node type consistency, content validation, and structural integrity to ensure the parsed tree accurately represents the intended document structure.

Performance debugging: The specialized debugging focused on algorithmic complexity, resource usage patterns, memory allocation behavior, and execution time analysis to identify and resolve performance bottlenecks in parsing operations.

Context dependency debugging: The debugging approach for parser rules that depend on surrounding content, document state, or position within other elements, requiring careful analysis of parsing context and state propagation.

Edge case handling: The systematic approach to defining consistent behavior for ambiguous input patterns, unusual syntax combinations, and boundary conditions that may not be explicitly covered by the CommonMark specification.

Architecture and Extension Terminology

These terms describe the extensibility mechanisms, architectural patterns, and design approaches that enable customization and enhancement of the core markdown renderer while maintaining system integrity.

Plugin architecture: The extensibility system that allows custom functionality to be added to the markdown renderer without modifying core components, including plugin registration, lifecycle management, and integration points for custom behavior.

Extension points: The defined integration interfaces where plugins can hook into the parsing or rendering pipeline, including custom block parsers, inline element handlers, HTML renderers, and output format generators.

Renderer plugins: The specific type of plugin that generates custom output formats or modifies HTML generation behavior for specific node types, enabling support for different target formats while reusing the parsing pipeline.

Plugin lifecycle: The managed states and transitions for plugin loading and cleanup, including `UNLOADED`, `LOADING`, `LOADED`, `ACTIVE`, `ERROR`, and `UNLOADING` states with defined transitions and error handling.

Compositional extensions: The design approach where new functionality is added through composition of existing components rather than inheritance or modification of core classes, enabling flexible customization while maintaining system stability.

CommonMark extensions: The standardized additions to the base CommonMark specification, including tables, task lists, strikethrough text, and other commonly supported features that extend basic markdown capabilities.

Pipeline architecture: The unidirectional data flow design where input is transformed through a sequence of processing stages, with each stage having well-defined inputs, outputs, and responsibilities that enable component isolation and testing.

Component isolation: The design principle ensuring clear separation between parsing logic components, with minimal dependencies and well-defined interfaces that enable independent development, testing, and modification of system parts.

Regular Expression and Pattern Matching Terminology

These terms describe the specific patterns, matching techniques, and regex-based parsing approaches used throughout the system for recognizing markdown syntax and extracting structured information.

ATX_HEADING_PATTERN: The regular expression pattern for matching hash-prefix headings, typically `^(#{1,6})\s+(.+?)(?:\s+#+)?$`, which captures the heading level through hash count and the heading text while allowing optional trailing hashes.

SETEXT_H1_UNDERLINE and **SETEXT_H2_UNDERLINE**: Regular expression patterns for matching Setext heading underlines, using patterns like `^=+$` for level 1 headings and `^-+$` for level 2 headings, requiring coordination with the previous line's content.

FENCED_CODE_START and **FENCED_CODE_END**: Patterns for matching the opening and closing delimiters of fenced code blocks, typically `^```(\w*)$` or `^~~~(\w*)$` for opening (with optional language specification) and corresponding closing patterns.

HORIZONTAL_RULE_PATTERN: The pattern for matching horizontal rules, such as `^(?:(?:\* *){3,}|(?:- *){3,}|(?:_ *){3,})$`, which recognizes three or more asterisks, dashes, or underscores with optional spacing.

BLOCKQUOTE_PATTERN: The pattern for matching blockquote lines, typically `^> ?(.*)$`, which captures the content after the blockquote marker while allowing for optional space after the greater-than symbol.

UNORDERED_MARKER_PATTERN and **ORDERED_MARKER_PATTERN**: Patterns for detecting list markers, such as `^( *)([*+-]) +(.*)$` for unordered lists and `^( *)(\d{1,9})[.)] +(.*)$` for ordered lists, capturing indentation, marker, and content.

HTML Generation and Output Terminology

These terms describe the specific concepts, techniques, and standards used when converting the parsed AST into HTML output, including escaping, formatting, and validation considerations.

HTML_ESCAPE_TABLE: The mapping data structure that defines character-to-entity conversions for HTML escaping, typically including `&` → `&`, `<` → `<`, `>` → `>`, `"` → `&quot;`, and `'` → `&#39;`.

ATTRIBUTE_ESCAPE_TABLE: The specialized character mapping for escaping HTML attribute values, which may have different escaping requirements than HTML content, particularly for quote characters and attribute-specific special characters.

VOID_ELEMENTS: The set of HTML5 element names that cannot have content or closing tags, including `area`, `base`, `br`, `col`, `embed`, `hr`, `img`, `input`, `link`, `meta`, `param`, `source`, `track`, and `wbr`.

Self-closing tags: HTML elements that don't have separate closing tags and are written as single tags like `<hr>` or `<img src="..." alt="...">`, requiring special handling during HTML generation to avoid generating incorrect closing tags.

Double-escaping: The error condition where HTML special characters are escaped multiple times, resulting in output like `&lt;` instead of `<`, typically caused by applying escaping at multiple stages of the processing pipeline.

Attribute value escaping: The specialized escaping rules for content within HTML attribute values, which must handle quote characters appropriately and may require different escape sequences than general HTML content escaping.

Visitor pattern: The design pattern used for traversing and operating on AST trees, where different node types accept visitor objects that implement specific operations, enabling separation of tree structure from processing algorithms.

Performance and Optimization Terminology

These terms describe the concepts and techniques related to parser performance, resource usage optimization, and scalability considerations in document processing systems.

Verbatim content blocks: Regions of the document (like code blocks) where content is preserved exactly as written without any markdown parsing, requiring specialized handling to avoid unnecessary processing while maintaining exact character preservation.

Algorithmic complexity: The analysis of parser operations in terms of time and space complexity, particularly important for operations like delimiter matching, list nesting resolution, and AST traversal that can exhibit quadratic or worse behavior on

pathological input.

Memory allocation patterns: The analysis of object creation and memory usage during parsing, including strategies for reducing garbage collection pressure, reusing temporary objects, and managing memory-intensive operations like large document processing.

Pipeline optimization: The techniques for improving performance across the entire parsing pipeline, including lazy evaluation, early termination conditions, and optimization of data structure choices for specific access patterns.

Input size scaling: The analysis of how parser performance changes with document size, including identification of operations that scale poorly and implementation of strategies to maintain reasonable performance on large documents.

This comprehensive glossary provides the terminological foundation for understanding, implementing, and extending markdown parsing systems. Each term represents a specific concept with precise meaning within the domain of document transformation, enabling clear communication about complex parsing algorithms and architectural decisions. The terminology spans from fundamental parsing concepts through specific implementation details to advanced extensibility patterns, supporting developers at all levels of expertise in building robust and maintainable markdown rendering systems.

Implementation Guidance

The terminology and concepts defined in this glossary should be implemented through consistent naming conventions and clear documentation standards throughout the codebase. This implementation guidance provides practical approaches for maintaining terminological consistency and supporting developer understanding.

Technology Recommendations for Documentation

Aspect	Simple Option	Advanced Option
Code Documentation	Inline comments with term definitions	Sphinx/godoc with cross-references
API Documentation	README with terminology section	OpenAPI specs with consistent terminology
Type Documentation	Docstrings using exact term names	Generated docs with glossary integration
Error Messages	Clear messages using standard terms	Structured errors with terminology links

Recommended Documentation Structure

```
project-root/
  docs/
    glossary.md              ← this comprehensive glossary
    api-reference.md         ← API docs using consistent terms
    architecture-guide.md    ← high-level concepts with term links
  src/
    types.py                 ← type definitions with docstring references
    constants.py             ← all named constants from glossary
    exceptions.py            ← error types with standard terminology
    utils/
      validation.py          ← terminology validation utilities
```

Terminology Validation Utilities

```python
# Complete terminology validation infrastructure

class TerminologyValidator:

    """Validates that code uses consistent terminology from the glossary."""



    def __init__(self, glossary_terms: Dict[str, str]):

        self.glossary_terms = glossary_terms

        self.deprecated_terms = {

            'markdown_node': 'ASTNode',

            'text_element': 'InlineNode',

            'block_element': 'BlockNode'

        }



    def validate_variable_names(self, source_code: str) -> List[str]:

        """Validate that variable names use standard terminology."""

        # TODO: Parse source code and extract variable names

        # TODO: Check against glossary_terms dictionary

        # TODO: Flag usage of deprecated_terms

        # TODO: Suggest corrections for non-standard terminology

        # TODO: Return list of validation errors with line numbers

        pass



    def validate_function_signatures(self, module) -> List[str]:

        """Validate that function names and parameters use standard terms."""

        # TODO: Inspect module functions using reflection

        # TODO: Check parameter names against standard terminology

        # TODO: Validate return type annotations use correct type names

        # TODO: Flag functions that don't follow naming conventions

        pass



# Standard constants with exact names from glossary
```

```
HTML_ESCAPE_TABLE = {

    '&': '&amp;',

    '<': '&lt;',

    '>': '&gt;',

    '"': '&quot;',

    "'": '&#39;'

}


ATTRIBUTE_ESCAPE_TABLE = {

    '&': '&amp;',

    '<': '&lt;',

    '>': '&gt;',

    '"': '&quot;',

    "'": '&#39;',

    '\n': '&#10;',

    '\r': '&#13;',

    '\t': '&#9;'

}


VOID_ELEMENTS = {

    'area', 'base', 'br', 'col', 'embed', 'hr', 'img',

    'input', 'link', 'meta', 'param', 'source', 'track', 'wbr'

}


ESCAPABLE_CHARS = '\\`*_{}[]()#+-.!|~'


EMPHASIS_DELIMITER_CHARS = {'*', '_'}
```

Core Terminology Integration

```python
def validate_ast_terminology(root: ASTNode) -> List[ValidationError]:
    """Validate AST structure uses standard terminology and relationships."""
    # TODO: Check that node_type values match NodeType enum
    # TODO: Verify block_type values match BlockType enum
    # TODO: Validate inline_type values match InlineType enum
    # TODO: Ensure parent-child relationships are properly established
    # TODO: Check that node attributes use standard attribute names
    # TODO: Return comprehensive validation errors with fix suggestions
    pass

def generate_terminology_report(codebase_path: str) -> str:
    """Generate report on terminology usage consistency across codebase."""
    # TODO: Scan all Python files in codebase_path
    # TODO: Extract type names, variable names, function names
    # TODO: Compare against standard terminology from glossary
    # TODO: Identify inconsistencies and deprecated usage
    # TODO: Generate formatted report with recommendations
    # TODO: Include metrics on terminology compliance percentage
    pass
```

Language-Specific Terminology Hints

For Python implementation:

- Use exact enum names from glossary: `NodeType.DOCUMENT`, `BlockType.PARAGRAPH`, `InlineType.STRONG`
- Follow PEP 8 naming: `parse_blocks()`, `render_to_html()`, `escape_html()`
- Use descriptive variable names: `ast_root` not `root`, `block_node` not `node`
- Include type hints with exact type names: `def parse_blocks(text: str) -> ASTNode`

For error messages:

- Use standard terminology: "Failed to parse ATX heading" not "Failed to parse hash heading"
- Reference specific types: "Expected BlockType.HEADING" not "Expected heading type"
- Include glossary term definitions in extended error messages for learning

Milestone Checkpoint: Terminology Consistency

After implementing terminology standards:

- Run `python -m terminology_validator src/` to check naming consistency

- Verify all type definitions match glossary exactly: `NodeType`, `BlockType`, `InlineType`
- Confirm error messages use standard terms from glossary
- Test that API documentation uses consistent terminology throughout
- Validate that code comments reference correct technical terms

Signs of terminology problems:

- Mixed naming conventions (some camelCase, some snake_case for same concepts)
- Error messages using informal terms not in glossary
- Type names that don't match the standard definitions
- Function parameters with non-standard names like `md_text` instead of `markdown_text`

Debugging Terminology Issues

Symptom	Likely Cause	How to Diagnose	Fix
Confusing error messages	Inconsistent terminology usage	Check error message text against glossary	Update messages to use standard terms
Hard to navigate codebase	Non-standard naming conventions	Audit variable/function names	Rename to match glossary conventions
Documentation inconsistencies	Multiple terms for same concept	Cross-reference docs with glossary	Standardize on single term per concept
Integration problems	Different components use different terms	Check inter-component interfaces	Align all interfaces to standard terminology

This implementation guidance ensures that the comprehensive terminology defined in the glossary is consistently applied throughout the codebase, enabling clear communication, easier maintenance, and better developer experience when working with the markdown renderer system.