Diff Tool: Design Document

Overview

A text comparison tool that identifies differences between two files using the Longest Common Subsequence algorithm and Myers' diff algorithm to generate unified diff output. The key architectural challenge is efficiently computing optimal edit sequences using dynamic programming while providing readable output formats.

> 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):** All milestones — understanding the fundamental problem drives all implementation decisions

Text comparison appears deceptively simple on the surface. Given two files, we want to identify what changed — which lines were added, deleted, or remained the same. However, this seemingly straightforward task reveals deep complexity when we consider the numerous ways text can be modified and the challenge of presenting changes in a meaningful, readable format.

The core challenge lies in defining what constitutes the "best" representation of differences between two text sequences. When a user edits a document, they don't simply perform random insertions and deletions — they make logical changes like adding paragraphs, moving sections, or revising sentences. Our diff algorithm must reverse-engineer these intentions from the final result, identifying the most intuitive way to express the transformation from one version to another.

Consider a simple example where we compare two versions of a function. The original version might have five lines, and the modified version might have seven lines. There could be multiple valid ways to represent this change: we could show that two lines were added at the end, or that the entire function was deleted and replaced, or that specific lines were inserted at particular positions. The algorithm must choose the representation that minimizes cognitive load for the human reader while accurately capturing the semantic changes.

The problem becomes even more complex when we consider that text files can contain thousands of lines, use different encodings, have varying line ending conventions, and may share very few common elements. A naive approach that compares every character or line against every other character or line quickly becomes computationally prohibitive. We need algorithms that can efficiently find optimal solutions while handling edge cases gracefully.

Mental Model: DNA Sequence Alignment

The most intuitive way to understand text comparison is through the lens of **DNA sequence alignment**, a fundamental problem in bioinformatics. Just as biologists compare genetic sequences to understand evolutionary relationships and identify mutations, we compare text files to understand editorial changes and identify modifications.

In DNA sequence alignment, scientists start with two genetic sequences — strings of nucleotides represented as letters (A, T, G, C). They want to find the optimal alignment that maximizes matching bases while minimizing the number of insertions, deletions, and substitutions needed to transform one sequence into another. The key insight is that not all differences are equally meaningful — some represent genuine evolutionary changes, while others might be artifacts of sequencing errors or irrelevant variations.

Similarly, when comparing text files, we have two sequences of lines (instead of nucleotides). Our goal is to find the optimal alignment that maximizes matching lines while minimizing the number of insertions and deletions needed to transform the first file into the second. Just as DNA alignment algorithms identify the longest matching subsequences to infer evolutionary relationships, diff algorithms identify the **longest common subsequence** of lines to infer editorial relationships.

> **Key Insight**: The longest common subsequence represents the "evolutionary backbone" — the content that remained stable between versions. Everything else represents editorial mutations that need to be highlighted.

This biological analogy illuminates several important aspects of the diff problem. First, **order matters** — just as the sequence of nucleotides in DNA determines genetic function, the sequence of lines in a text file determines logical structure. We can't arbitrarily rearrange lines to maximize matches without losing semantic meaning. Second, **context is crucial** — just as biologists present alignments with surrounding sequences to help interpret the significance of mutations, diff tools must present changes with surrounding context lines to help readers understand the scope and impact of modifications.

Third, **optimal alignment requires global perspective** — we can't make locally optimal decisions about matching lines without considering the global structure. A line that appears in both files might seem like an obvious match, but if aligning those lines forces many other lines to appear as insertions and deletions, a different alignment might produce a cleaner, more readable result.

The DNA analogy also reveals why this problem is computationally challenging. With sequences of length m and n, there are exponentially many possible alignments to consider. Dynamic programming algorithms like those used in bioinformatics provide a systematic way to explore all possibilities efficiently, building up optimal solutions for smaller subsequences and combining them to solve the larger problem.

Existing Diff Algorithms

The evolution of diff algorithms reflects a progression from simple, intuitive approaches to sophisticated techniques that balance computational efficiency with output quality. Understanding this progression helps us appreciate the design decisions in modern diff tools and choose the right algorithm for our specific requirements.

Decision: Algorithm Selection Strategy

- **Context**: Need to choose between multiple diff algorithms with different performance and quality characteristics
- **Options Considered**: Naive character comparison, line-by-line comparison, LCS-based approach, Myers' algorithm
- **Decision**: Implement LCS-based approach first, with Myers' algorithm as future enhancement
- **Rationale**: LCS provides optimal results for learning dynamic programming concepts, while Myers' offers better performance for large files
- **Consequences**: Excellent output quality but O(mn) space complexity; Myers' extension will require significant architectural changes

Naive Character-by-Character Comparison

The most straightforward approach treats files as sequences of characters and attempts to find character-level differences. This algorithm scans through both files simultaneously, marking characters as matching or different based on direct comparison.

The naive character approach works by maintaining two pointers, one for each file, and advancing them in lockstep. When characters match, both pointers advance. When characters differ, the algorithm faces a decision: should it treat this as a substitution, or should it look ahead to see if one file has an insertion or deletion at this position?

The fundamental problem with character-level comparison is **alignment ambiguity**. Consider comparing "abcde" with "axbcde". The naive algorithm might identify 'a' as matching, then encounter 'b' versus 'x' and incorrectly conclude that 'b' was substituted with 'x', 'c' was substituted with 'b', and so on. The correct interpretation — that 'x' was inserted between 'a' and 'b' — requires lookahead or backtracking capabilities that the naive approach lacks.

Aspect	Character-by-Character Approach
Time Complexity	O(n) for simple scan, O(nm) with backtracking
Space Complexity	O(1) for simple scan, O(nm) for optimal alignment
Output Quality	Poor - many false substitutions, misaligned content
Implementation Complexity	Very simple for basic version, complex for quality results
Use Cases	Binary file comparison, simple text validation

Despite these limitations, character-level comparison remains valuable for specific scenarios. Binary file comparison often uses this approach because line-based comparison is meaningless for non-text data. Additionally, character-level diffs can provide fine-grained analysis within lines that have been identified as changed by higher-level algorithms.

Line-by-Line Comparison

Recognizing that most text editing operations work at the line level, line-by-line comparison treats files as sequences of line strings rather than character sequences. This approach significantly reduces the search space and produces more semantically meaningful results for typical text files.

Line-based comparison begins by tokenizing both files into arrays of lines, preserving the original line structure including empty lines. The algorithm then applies sequence comparison techniques to these line arrays, treating each line as an atomic unit. This abstraction level matches how humans typically think about text editing — we add paragraphs, delete sections, and modify sentences, rather than randomly inserting and deleting characters.

The line-based approach immediately solves several problems that plague character-level comparison. **Alignment ambiguity** is greatly reduced because entire lines serve as strong anchors for alignment. **Semantic coherence** improves because the algorithm naturally preserves logical text structure. **Performance** increases dramatically because we're comparing hundreds or thousands of lines instead of millions of characters.

However, line-based comparison introduces its own challenges. **Whitespace sensitivity** can cause semantically identical lines to appear different due to trailing spaces or tab-versus-space differences. **Line ending variations** (LF, CRLF, CR) can create spurious differences between files created on different operating systems. **Granularity limitations** mean that changes within a line appear as complete line replacements, even if only a single word was modified.

Aspect	Line-by-Line Approach
Time Complexity	O(n) for simple scan, depends on subsequence algorithm for quality results
Space Complexity	O(n + m) for line storage plus algorithm-specific requirements
Output Quality	Good - matches human editing patterns, preserves document structure
Implementation Complexity	Moderate - requires tokenization, normalization, and subsequence algorithms
Use Cases	Most text files, source code, configuration files, documentation

Modern diff tools universally adopt line-based comparison as their foundation, then apply various algorithms to find optimal line sequence alignments. The choice of subsequence algorithm determines the tool's performance characteristics and output quality.

Longest Common Subsequence (LCS) Approach

The **Longest Common Subsequence** algorithm provides a rigorous mathematical foundation for optimal sequence comparison. Unlike heuristic approaches, LCS guarantees finding the longest sequence of lines that appear in both files in the same order, providing a principled way to identify what remained unchanged between versions.

LCS operates on the insight that the optimal diff should maximize the amount of content marked as "unchanged" while minimizing the content marked as "added" or "deleted". By finding the longest possible common subsequence, we identify the maximum amount of shared content, and everything else naturally falls into insertion or deletion categories.

The algorithm uses **dynamic programming** to build up optimal solutions systematically. It constructs a two-dimensional matrix where entry (i,j) represents the length of the LCS between the first i lines of file A and the first j lines of file B. The recurrence relation captures the essential logic: if lines i and j match, we can extend the LCS from position (i-1, j-1); otherwise, we take the better of the two options where one file advances and the other doesn't.

This mathematical rigor comes with computational costs. The classic LCS algorithm requires O(mn) time and O(mn) space, where m and n are the lengths of the input sequences. For large files, this can consume substantial memory. However, the algorithm's guarantee of optimal results makes it an excellent choice for learning dynamic programming concepts and understanding the theoretical foundation of diff algorithms.

Aspect	LCS-Based Approach
Time Complexity	O(mn) - must consider all position pairs
Space Complexity	O(mn) for full matrix, O(min(m,n)) with optimization
Output Quality	Optimal - guaranteed to find longest common subsequence
Implementation Complexity	Moderate - requires understanding of dynamic programming
Use Cases	Educational purposes, small to medium files, maximum quality required

The LCS approach serves as an excellent foundation for understanding diff algorithms because it makes the optimization goal explicit and provides a clear algorithmic framework. Once developers understand how LCS works, they can appreciate the trade-offs made by more advanced algorithms.

Myers' Diff Algorithm

Myers' algorithm represents the state-of-the-art in diff algorithm design, providing the same optimal results as LCS while offering significantly better performance characteristics for typical use cases. Developed by Eugene Myers in 1986, this algorithm powers many modern version control systems and diff tools.

Myers' key insight is that most real-world file comparisons involve relatively few changes compared to the total file size. While the worst-case scenario still requires O(mn) time, the expected case for files with few differences runs in $O(n + d^2)$ time, where d is the number of differences. This makes the algorithm extremely fast for the common case of comparing similar files with localized changes.

The algorithm reconceptualizes the diff problem as a **graph traversal** problem. Instead of building a complete dynamic programming matrix, Myers' algorithm explores only the portions of the search space that are likely to contain optimal solutions. It uses a clever **edit graph** representation where horizontal moves represent insertions, vertical moves represent deletions, and diagonal moves represent matches.

The graph traversal proceeds in **waves**, exploring all possible paths that require exactly k edit operations before exploring paths that require k+1 operations. This breadth-first approach guarantees that the first path to reach the end represents an optimal solution, allowing the algorithm to terminate early without exploring the entire search space.

Aspect	Myers' Algorithm
Time Complexity	O(n + d²) expected case, O(mn) worst case
Space Complexity	O(d) for the search frontier
Output Quality	Optimal - same results as LCS but computed more efficiently
Implementation Complexity	High - requires understanding of graph algorithms and optimization techniques
Use Cases	Production diff tools, version control systems, large file comparison

Myers' algorithm also provides natural opportunities for further optimizations. The **linear space** variant reduces memory usage to O(d), making it practical for comparing very large files. **Patience diff** and **histogram diff** build on Myers' foundation to handle common patterns like code movement and repeated sections more elegantly.

Algorithm Comparison and Selection Criteria

The choice between diff algorithms depends on multiple factors including file size, expected difference density, performance requirements, implementation complexity, and output quality needs. Understanding these trade-offs helps architects make informed decisions for their specific use cases.

Algorithm	Best For	Avoid When	Key Advantage	Major Limitation
Naive Character	Binary files, simple validation	Text files, quality matters	Simple implementation	Poor alignment quality
Line-by-Line Heuristic	Quick prototypes, approximations	Optimal results required	Fast implementation	Suboptimal results
LCS Dynamic Programming	Learning, guaranteed optimal results	Large files, memory constrained	Theoretical foundation	High memory usage
Myers' Algorithm	Production systems, large files	Educational contexts	Best practical performance	Implementation complexity

> **Architecture Insight**: The progression from naive approaches to Myers' algorithm illustrates a fundamental pattern in algorithm design — starting with simple, correct solutions and then optimizing based on real-world usage patterns and constraints.

For our diff tool implementation, we'll begin with the LCS approach because it provides the clearest path for learning dynamic programming concepts while guaranteeing optimal results. The mathematical rigor of LCS helps developers understand exactly what "optimal" means in the context of sequence comparison, providing a solid foundation for appreciating the optimizations made by more advanced algorithms.

The modular architecture we'll design can accommodate future algorithm upgrades. By separating the sequence comparison logic from file handling and output formatting, we can later swap in Myers' algorithm or other

approaches without affecting the rest of the system. This extensibility principle ensures that our learning-focused implementation can evolve into a production-quality tool.

Implementation Guidance

The context and problem understanding phase requires no code implementation, but establishing the right mental models and terminology now will guide all subsequent development decisions. This section provides the conceptual foundation and research direction for implementing a robust diff tool.

A. Research and Analysis Recommendations

Activity	Beginner Approach	Advanced Exploration
Algorithm Study	Implement LCS by hand on paper with 3-4 line examples	Study Myers' paper and implement edit graph traversal
Tool Analysis	Run `diff -u` on sample files, analyze output format	Compare `diff`, `git diff`, and specialized tools like `wdiff`
Performance Testing	Time algorithms on files with 100-1000 lines	Benchmark with real codebases, measure memory usage
Format Research	Study unified diff format specification	Explore side-by-side, context, and HTML output formats

B. Conceptual Validation Exercises

Before writing any code, validate your understanding of the core concepts through these exercises:

Exercise 1: Manual LCS Computation Take two short text sequences (5-8 lines each) and manually compute their LCS using the dynamic programming matrix. This hands-on experience will reveal common indexing pitfalls and help you understand the backtracking process.

Example sequences for practice:

```
File A: ["apple", "banana", "cherry", "date"]
File B: ["banana", "cherry", "elderberry", "fig"]
```

Exercise 2: Diff Format Analysis Create several test file pairs and run them through standard `diff -u` command. Analyze the output format, paying attention to:

- How line numbers are calculated and displayed
- Where context lines appear and how many are included
- How consecutive changes are grouped into hunks
- What happens with edge cases like empty files or no common lines

Exercise 3: Algorithm Comparison For the same file pair, manually trace through different algorithmic approaches:

- Naive character-by-character (identify where it fails)
- Simple line matching without LCS (show suboptimal results)
- LCS-based approach (demonstrate optimal alignment)

C. Key Terminology and Concepts

Establish consistent vocabulary that will be used throughout the implementation:

Term	Definition	Example
Sequence	Ordered collection of comparable elements	Array of lines from a text file
Common Subsequence	Elements appearing in both sequences in same order	Lines present in both file versions
Edit Distance	Minimum operations to transform one sequence to another	Number of insertions + deletions needed
Hunk	Group of consecutive changes with surrounding context	Block of diff output starting with @@
Context Lines	Unchanged lines displayed around changes for readability	Lines before/after changes in diff output

D. Problem Decomposition Strategy

Break down the overall diff problem into manageable subproblems that map to our milestone structure:

1. **File Representation Problem**: How do we convert file bytes into comparable line sequences while handling encoding and line ending variations?
2. **Sequence Alignment Problem**: How do we find the optimal way to align two line sequences to minimize apparent changes?
3. **Change Classification Problem**: How do we convert the alignment result into explicit add/delete/unchanged operations?
4. **Output Formatting Problem**: How do we present the change operations in a readable, standard format?
5. **User Interface Problem**: How do we provide command-line access with appropriate options and error handling?

E. Design Principles for Implementation

Establish core principles that will guide implementation decisions:

> **Principle 1: Correctness Over Performance** Prioritize producing correct, optimal results over algorithmic optimizations. Performance improvements can come later without changing the external interface.

> **Principle 2: Modularity for Learning** Design each component to be independently testable and understandable. A learner should be able to focus on one algorithmic concept at a time.

> **Principle 3: Standard Compatibility** Produce output that matches established diff tool conventions. This allows comparison with reference implementations for validation.

> **Principle 4: Graceful Degradation** Handle edge cases and errors gracefully, providing helpful error messages rather than crashing or producing incorrect output.

F. Milestone Checkpoint Expectations

After completing this context and problem statement analysis, you should be able to:

- ☐ **Explain the LCS concept** using the DNA alignment analogy to someone unfamiliar with algorithms
- ☐ **Manually compute** the LCS for simple 4-5 line file examples using the dynamic programming matrix
- ☐ **Identify the trade-offs** between naive, LCS-based, and Myers' algorithmic approaches
- ☐ **Analyze diff output** from standard tools, understanding the unified format structure
- ☐ **Decompose the overall problem** into the specific subproblems addressed by each milestone

G. Common Conceptual Pitfalls

⚠️ **Pitfall: Confusing LCS with String Edit Distance** LCS finds the longest common subsequence, while edit distance finds the minimum number of operations needed for transformation. These are related but different concepts. LCS focuses on what stays the same; edit distance focuses on what changes.

⚠️ **Pitfall: Assuming Character-Level Comparison is Simpler** Character-level comparison seems simpler but produces poor results for text files. Line-level comparison is actually easier to implement correctly and produces much better output.

⚠️ **Pitfall: Underestimating Memory Requirements** The naive LCS algorithm requires O(mn) space, which can be substantial for large files. A 10,000-line file comparison needs 100 million matrix entries, potentially several gigabytes of memory.

⚠️ **Pitfall: Ignoring File Encoding Issues** Text files can use various encodings (UTF-8, Latin-1, ASCII) and line endings (LF, CRLF, CR). Ignoring these differences can cause identical files to appear different or cause decoding errors.

This conceptual foundation prepares you for the detailed architectural design and implementation phases. The mental models and terminology established here will guide decision-making throughout the development process, ensuring that implementation details serve the larger goal of creating an intuitive, correct, and efficient diff tool.

Goals and Non-Goals

> **Milestone(s):** All milestones — clearly defining scope prevents feature creep and guides implementation decisions throughout the project

Setting clear boundaries for our diff tool implementation is crucial for maintaining focus and delivering a working solution within reasonable complexity. This section establishes what we will build, what we explicitly won't build, and the rationale behind these decisions. By defining these boundaries upfront, we can make consistent architectural decisions and avoid the temptation to add features that would complicate the core learning objectives.

The scope definition serves multiple purposes beyond project management. It helps us choose the right algorithms and data structures for our specific use case, guides our testing strategy by defining the expected behavior boundaries, and sets realistic expectations for performance and functionality. Most importantly for a learning project, it keeps us focused on the core concepts of dynamic programming, edit distance computation, and diff algorithm implementation without getting distracted by peripheral concerns.

Functional Goals

Our diff tool will implement a focused set of core capabilities that demonstrate the fundamental concepts of text comparison algorithms while providing practical utility. These goals are designed to cover the essential user workflows for comparing text files while maintaining implementation simplicity.

Mental Model: Essential Workshop Tools — Think of our functional goals like selecting tools for a woodworking workshop. We're not trying to build every possible tool, but rather the essential ones that can handle 80% of common tasks well. A good plane, saw, and chisel will accomplish most woodworking projects, even if specialized tools might be marginally better for specific tasks. Similarly, our diff tool focuses on the core comparison operations that handle the vast majority of real-world text comparison needs.

Core Text Comparison Capabilities

The foundation of our tool is reliable text comparison between two files using the Longest Common Subsequence algorithm. This provides the theoretical correctness that many users expect from a diff tool — finding the optimal sequence of changes that transforms one file into another.

Capability	Implementation Approach	Success Criteria
Line-based comparison	Split files into line arrays, compare using LCS	Identical results to reference implementations
UTF-8 and Latin-1 encoding support	Automatic encoding detection with fallback	Handles common text file encodings without corruption
Multiple line ending formats	Normalize LF, CRLF, and CR during tokenization	Consistent behavior across Windows, Unix, and Mac files
Empty file handling	Special case detection with appropriate output	Graceful handling without algorithm failures
Large file support	Streaming line reading with memory management	Process files up to several MB without memory exhaustion

The line-based approach aligns with how humans naturally think about text changes — additions, deletions, and modifications typically happen at the granularity of lines rather than individual characters. This choice also significantly reduces the computational complexity compared to character-level diffing while still providing useful results for most text files.

Unified Diff Format Output

Our output format will conform to the unified diff standard, ensuring compatibility with existing tools and user expectations. The unified format strikes an optimal balance between human readability and machine parseability.

Format Element	Structure	Example
File headers	`--- filename1` and `+++ filename2`	`--- original.txt`
Hunk headers	`@@ -start,count +start,count @@`	`@@ -15,7 +15,9 @@`
Context lines	Lines without prefix	`unchanged line`
Deletion markers	Lines prefixed with `-`	`- deleted content`
Addition markers	Lines prefixed with `+`	`+ added content`

The unified format provides several advantages over alternatives like side-by-side or context diff formats. It's compact for large changes, clearly shows the relationship between deletions and additions, and is universally supported by version control systems, patch utilities, and code review tools.

Configurable Context Display

Context lines around changes provide crucial readability for understanding the location and nature of modifications. Our implementation will support configurable context amounts to balance between providing sufficient context and keeping output concise.

Decision: Default Context Lines

- **Context**: Users need context to understand where changes occur, but too much context clutters output
- **Options Considered**: 0 lines (minimal), 3 lines (git default), 5 lines (traditional diff)
- **Decision**: Default to 3 context lines with configurable override
- **Rationale**: Matches git behavior for user familiarity, provides sufficient context for most changes, keeps output manageable for review
- **Consequences**: Enables intuitive usage while allowing customization for specific needs

Context Setting	Use Case	Output Characteristics
0 lines	Minimal output, large files	Only changed lines shown
1-2 lines	Quick scanning	Limited context, compact
3 lines (default)	Code review, general use	Good balance of context and brevity
5+ lines	Understanding complex changes	Extensive context, verbose output

Command-Line Interface

The CLI provides the primary user interaction with our diff tool, emphasizing simplicity and following Unix conventions for predictable behavior.

Interface Element	Specification	Rationale
Basic invocation	`difftool file1 file2`	Matches standard diff utility patterns
Context control	`--context N` or `-c N`	Standard flag naming for context lines
Color control	`--no-color`	Disable ANSI codes for piping/redirection
Help output	`--help` or `-h`	Standard help flag conventions
Exit codes	0 (identical), 1 (different), 2 (error)	Scriptable behavior matching diff standards

The interface design prioritizes discoverability and consistency with existing tools. Users familiar with standard diff utilities should be able to use our tool immediately without consulting documentation for basic operations.

Visual Enhancement Features

Color output significantly improves the readability of diff results in terminal environments by providing immediate visual cues about change types.

Enhancement	Implementation	Benefit
Colored additions	Green text for `+` lines	Immediate recognition of new content
Colored deletions	Red text for `-` lines	Clear identification of removed content
TTY detection	Automatic color disabling for non-terminals	Prevents ANSI codes in files/pipes
Color override	Manual disable via flag	User control for accessibility needs

The color scheme follows conventional diff tool coloring that users expect, reducing cognitive load when interpreting results. Automatic TTY detection ensures that color codes don't interfere with file output or shell scripting scenarios.

Non-Goals

Explicitly defining what we will not implement is equally important as defining what we will build. These exclusions keep our scope manageable while focusing on the core learning objectives around dynamic programming and diff algorithms.

Mental Model: Focused Learning Path — Think of non-goals like choosing a hiking trail. When learning to hike, you pick a trail that matches your current skill level and available time. You don't attempt advanced mountaineering techniques or multi-day expeditions on your first outing. Similarly, our non-goals represent advanced features that would distract from mastering the fundamental concepts of text comparison algorithms.

Advanced Diff Algorithms

While Myers' algorithm offers superior performance characteristics, implementing it properly requires understanding edit graphs, diagonal traversal, and complex optimization techniques that go beyond the scope of a beginner project focused on dynamic programming concepts.

Excluded Algorithm	Complexity Reason	Alternative Learning Path
Myers' algorithm	Edit graph theory, O(ND) complexity analysis	Advanced algorithms course
Patience diff	Requires understanding of unique line identification	Specialized diff algorithm study
Histogram diff	Statistical analysis of file content patterns	Machine learning or statistics focus
Semantic diff	Language parsing, AST comparison	Compiler design or language processing

Our LCS-based approach provides the theoretical foundation that makes these advanced algorithms comprehensible later. Students who master the dynamic programming approach will be well-prepared to understand how Myers' algorithm optimizes the same underlying problem.

Multi-File and Directory Operations

Directory comparison introduces file system traversal, recursive algorithms, and complex output formatting that would overshadow the core diff algorithm learning objectives.

Excluded Feature	Implementation Complexity	Focus Dilution
Directory diffing	Recursive tree traversal, file matching	File system operations vs. algorithms
Batch file comparison	Process management, parallel execution	Concurrency vs. dynamic programming
Archive comparison	Compression format handling	File format parsing vs. text comparison
Remote file access	Network protocols, authentication	Distributed systems vs. local algorithms

These features require substantial infrastructure code that doesn't contribute to understanding text comparison algorithms. Students can add these capabilities later once they've mastered the core diff computation.

Binary File Support

Binary file comparison requires fundamentally different algorithms and output formats that don't align with our text-focused learning objectives.

> **Decision: Text Files Only**
>
> - **Context**: Binary files require byte-level comparison with different visualization needs
> - **Options Considered**: Hex dump diff, binary detection with error, universal byte comparison
> - **Decision**: Detect binary files and report error with helpful message
> - **Rationale**: Keeps focus on text algorithms, avoids complex binary visualization, provides clear user feedback
> - **Consequences**: Users must use specialized tools for binary comparison, but get clear guidance on limitation

Binary Scenario	Our Behavior	User Guidance
Pure binary files	Error with detection message	Suggest `cmp` or `xxd` utilities
Mixed text/binary	Attempt text processing, may show garbage	Warning about potential binary content
Large binary files	Early detection to avoid memory issues	Clear error before processing begins

Advanced Output Formats

Supporting multiple output formats would require complex formatting infrastructure that distracts from the algorithmic learning goals.

Excluded Format	Implementation Overhead	Learning Distraction
Side-by-side display	Terminal width detection, column alignment	UI formatting vs. algorithm implementation
HTML output	Web formatting, CSS styling, escaping	Web development vs. text processing
JSON/XML structured output	Schema definition, serialization libraries	Data formats vs. diff computation
Patch file generation	Advanced header formatting, metadata handling	File format specifications vs. algorithms

The unified diff format provides sufficient functionality for learning purposes while maintaining compatibility with existing tools. Students can explore alternative formats in follow-up projects once they understand the underlying comparison logic.

Three-Way Merge and Conflict Resolution

Merge algorithms introduce conflict detection, resolution strategies, and complex user interaction patterns that represent a separate area of study from basic two-way comparison.

Merge Feature	Algorithmic Complexity	UI Complexity
Three-way merge	Multiple LCS computations, conflict detection	Interactive conflict resolution
Automatic resolution	Heuristic development, edge case handling	Strategy configuration interfaces
Manual conflict editing	Text editor integration, temporary files	File manipulation, process management
Merge base detection	Version control concepts, graph algorithms	Repository integration requirements

Three-way merging builds upon two-way diffing but requires additional theoretical knowledge about merge bases, conflict categories, and resolution strategies. This complexity would double the scope of our project without proportional learning benefit for the core dynamic programming concepts.

Performance Optimization Features

Advanced performance optimizations require profiling tools, memory management techniques, and algorithmic analysis that extend beyond the introductory scope.

Optimization	Technical Requirements	Learning Prerequisites
Memory streaming	Chunk-based processing, temporary storage	Systems programming concepts
Parallel processing	Thread management, work distribution	Concurrency programming
Caching systems	Hash computation, cache invalidation	Performance engineering
Incremental diffing	Change detection, partial recomputation	Advanced algorithm optimization

Students benefit more from implementing a correct, straightforward algorithm first, then learning optimization techniques in dedicated performance-focused projects. The educational value comes from understanding the algorithmic complexity and correctness, not from premature optimization.

Implementation Guidance

The scope decisions outlined above directly influence our technology choices and code organization. This guidance provides concrete recommendations for translating these goals into a working implementation.

Technology Recommendations

Component	Simple Option	Advanced Option
File I/O	Built-in file operations (`open`, `read`)	Memory-mapped files (`mmap`)
Encoding Detection	Try UTF-8, fallback to Latin-1	`chardet` library with confidence scoring
CLI Parsing	Manual `sys.argv` processing	`argparse` with subcommands and validation
Color Output	Simple ANSI escape sequences	`colorama` for cross-platform support
Testing Framework	Built-in `unittest` module	`pytest` with fixtures and parameterization

For learning purposes, the simple options provide direct control and understanding of the underlying operations. Students can upgrade to advanced options in later iterations once they understand the basic functionality.

Recommended File Structure

Organizing code to match our functional goals helps maintain clear separation of concerns:

```
diff-tool/
├── src/
│   ├── __init__.py
│   ├── main.py                 # CLI entry point and argument parsing
│   ├── file_reader.py          # File I/O and line tokenization
│   ├── lcs_engine.py          # LCS algorithm implementation
│   ├── diff_generator.py      # Edit script and hunk formation
│   └── output_formatter.py    # Unified diff format generation
├── tests/
│   ├── test_file_reader.py    # File I/O edge cases
│   ├── test_lcs_engine.py     # Algorithm correctness
│   ├── test_diff_generator.py # Hunk formation logic
│   └── test_integration.py    # End-to-end scenarios
├── examples/
│   ├── sample1.txt           # Test files for manual verification
│   ├── sample2.txt
│   └── expected_output.diff
└── README.md                 # Usage instructions and examples
```

This structure isolates each major component in its own module, making it easier to test individual pieces and understand the data flow between components.

Scope Validation Checklist

Use this checklist during implementation to ensure you stay within the defined scope:

Functional Goals Checklist:

- [ ] Reads two text files specified as command-line arguments
- [ ] Handles UTF-8 and Latin-1 encoding automatically
- [ ] Normalizes different line ending formats (LF, CRLF, CR)
- [ ] Implements LCS algorithm with dynamic programming
- [ ] Generates unified diff format output
- [ ] Supports configurable context lines (default 3)
- [ ] Provides colored output in terminal environments
- [ ] Returns appropriate exit codes (0, 1, 2)
- [ ] Includes `--help` and `--no-color` flags

Non-Goals Validation:

- [ ] No directory comparison features
- [ ] No binary file processing (error with clear message)
- [ ] No side-by-side or HTML output formats
- [ ] No three-way merge capabilities
- [ ] No advanced performance optimizations
- [ ] No network or remote file access
- [ ] No Myers' algorithm implementation

Milestone Checkpoints

Each milestone should be validated against our scope boundaries:

Milestone 1 Checkpoint (Line Tokenization):

```bash
# Test basic functionality
python src/main.py examples/sample1.txt examples/sample2.txt

# Verify encoding handling
python src/main.py utf8_file.txt latin1_file.txt

# Check line ending normalization
python src/main.py windows_file.txt unix_file.txt
```

Expected behavior: Files read successfully, line counts reported, no encoding errors for supported formats, binary files rejected with clear error message.

Milestone 2 Checkpoint (LCS Algorithm):

```python
# Manual LCS verification
from src.lcs_engine import LCSEngine

engine = LCSEngine()
lines1 = ["A", "B", "C"]
lines2 = ["A", "X", "C"]
lcs = engine.compute_lcs(lines1, lines2)
assert lcs == ["A", "C"]  # Expected longest common subsequence
```

Expected behavior: LCS algorithm produces correct results for known inputs, handles empty files gracefully, completes in reasonable time for moderately sized files.

Milestone 3 Checkpoint (Diff Generation):

```bash
# Generate diff output
python src/main.py file1.txt file2.txt > output.diff

# Verify unified format
grep "^@@" output.diff  # Should show hunk headers
grep "^-" output.diff   # Should show deletions
grep "^+" output.diff   # Should show additions
```

Expected behavior: Output follows unified diff format, hunk headers show correct line numbers, context lines appear around changes, output is compatible with patch utilities.

Milestone 4 Checkpoint (CLI and Color):

```bash
# Test CLI options
python src/main.py --help                    # Shows usage information
python src/main.py --context 5 file1 file2  # Uses 5 context lines
python src/main.py --no-color file1 file2   # No ANSI codes in output

# Test exit codes
python src/main.py identical1.txt identical2.txt; echo $?  # Should print 0
python src/main.py different1.txt different2.txt; echo $?  # Should print 1
```

Expected behavior: Help text displays correctly, context option changes output, no-color flag removes ANSI codes, exit codes match file comparison results.

Common Scope Creep Pitfalls

⚠️ **Pitfall: Adding Word-Level Diffing** Beginning programmers often want to add word-level or character-level diffing when they see that line-level changes don't show small modifications clearly. This significantly increases algorithm complexity and output formatting requirements. Instead, focus on making the line-level algorithm robust and correct. Word-level diffing can be a follow-up project.

⚠️ **Pitfall: Implementing Directory Recursion** The temptation to add `diff -r` functionality is strong, but directory traversal introduces file system edge cases, symbolic link handling, and complex output organization. These concerns distract from the core algorithmic learning. Stick to two-file comparison.

⚠️ **Pitfall: Premature Performance Optimization** Students often worry about performance and want to add caching, parallel processing, or memory optimization before the basic algorithm works correctly. Focus on correctness first. Performance optimization requires profiling and measurement that's beyond the scope of learning dynamic programming concepts.

⚠️ **Pitfall: Over-Engineering the CLI** Rich command-line interfaces with subcommands, configuration files, and extensive options seem professional but require significant infrastructure code. Keep the CLI simple and focused on the core comparison operation. Advanced CLI features can be added in later iterations.

The key to successful scope management is recognizing when a feature request or implementation idea moves beyond the learning objectives. Ask yourself: "Does this help me understand dynamic programming and diff algorithms better, or does it distract from that goal?" Use that question to guide scope decisions throughout development.

Implementation Guidance

Technology Stack Recommendations

Component	Beginner Choice	Rationale
Language	Python 3.8+	Built-in file I/O, string handling, clear syntax for algorithm focus
CLI Framework	`argparse` module	Standard library, sufficient for simple flag handling
File I/O	Built-in `open()` with encoding parameter	Direct control over encoding, no external dependencies
Testing	`unittest` module	Standard library, familiar structure, good for learning TDD
Color Output	Manual ANSI escape codes	Understanding of terminal control, no dependencies

Essential Infrastructure Code

File Encoding Detection Helper:

```python
import codecs

def detect_file_encoding(filepath):
    """Detect file encoding, trying UTF-8 first, then Latin-1."""
    encodings = ['utf-8', 'latin-1']

    for encoding in encodings:
        try:
            with open(filepath, 'r', encoding=encoding) as f:
                f.read()  # Try to read entire file
            return encoding
        except UnicodeDecodeError:
            continue

    raise ValueError(f"Could not decode {filepath} with supported encodings")

def read_file_lines(filepath):
    """Read file lines with automatic encoding detection."""
    encoding = detect_file_encoding(filepath)

    with open(filepath, 'r', encoding=encoding, newline='') as f:
        content = f.read()

    # Normalize line endings and split
    content = content.replace('\r\n', '\n').replace('\r', '\n')
    lines = content.split('\n')

    # Handle trailing newline consistently
    if lines and lines[-1] == '':
```

```
        lines.pop()


    return lines
```

ANSI Color Helper:

```python
import sys


class ColorFormatter:
    """Handle colored output with automatic TTY detection."""


    RED = '\033[31m'
    GREEN = '\033[32m'
    RESET = '\033[0m'


    def __init__(self, use_color=None):
        if use_color is None:
            self.use_color = sys.stdout.isatty()
        else:
            self.use_color = use_color


    def red(self, text):
        """Format text in red for deletions."""
        if self.use_color:
            return f"{self.RED}{text}{self.RESET}"
        return text


    def green(self, text):
        """Format text in green for additions."""
        if self.use_color:
            return f"{self.GREEN}{text}{self.RESET}"
        return text
```

Core Component Skeletons

LCS Engine (for student implementation):

```python
class LCSEngine:
    """Implement Longest Common Subsequence using dynamic programming."""

    def compute_lcs(self, sequence1, sequence2):
        """
        Find the longest common subsequence between two sequences.

        Args:
            sequence1: List of comparable elements (typically lines)
            sequence2: List of comparable elements (typically lines)

        Returns:
            List containing the longest common subsequence elements
        """
        # TODO 1: Create DP table with dimensions (len(seq1)+1) x (len(seq2)+1)
        # TODO 2: Initialize first row and column to 0
        # TODO 3: Fill DP table using recurrence relation:
        #         if seq1[i-1] == seq2[j-1]: dp[i][j] = dp[i-1][j-1] + 1
        #         else: dp[i][j] = max(dp[i-1][j], dp[i][j-1])
        # TODO 4: Backtrack from dp[len(seq1)][len(seq2)] to recover actual LCS
        # TODO 5: Return the LCS elements in correct order

        pass  # Student implements this
```

Diff Generator (for student implementation):

```python
class DiffGenerator:
    """Convert LCS results into unified diff format hunks."""

    def generate_diff(self, lines1, lines2, lcs, context_lines=3):
        """
        Generate unified diff hunks from LCS computation.

        Args:
            lines1: Original file lines
            lines2: Modified file lines
            lcs: Longest common subsequence from LCS engine
            context_lines: Number of context lines around changes

        Returns:
            List of Hunk objects ready for formatting
        """
        # TODO 1: Create edit script marking each line as UNCHANGED, DELETED, or ADDED
        # TODO 2: Group consecutive changes together
        # TODO 3: Add context lines before and after each change group
        # TODO 4: Merge overlapping hunks when context causes overlap
        # TODO 5: Create Hunk objects with line ranges and content

        pass  # Student implements this
```

Milestone Validation Commands

Milestone 1 - File Reading:

```bash
# Create test files with different encodings
echo -e "Line 1\nLine 2\nLine 3" > test1.txt
echo -e "Line 1\nModified Line 2\nLine 3" > test2.txt

# Test basic file reading
python -c "
from src.file_reader import read_file_lines
lines = read_file_lines('test1.txt')
print(f'Read {len(lines)} lines')
for i, line in enumerate(lines): print(f'{i}: {repr(line)}')
"
```

Expected: Clean line reading without encoding errors, proper line ending normalization.

Milestone 2 - LCS Algorithm:

```bash
# Test LCS computation with known examples
python -c "
from src.lcs_engine import LCSEngine
engine = LCSEngine()

# Simple test case
seq1 = ['A', 'B', 'C', 'D']
seq2 = ['A', 'X', 'C', 'Y']
lcs = engine.compute_lcs(seq1, seq2)
print(f'LCS: {lcs}')  # Should be ['A', 'C']

# Empty sequence test
lcs_empty = engine.compute_lcs([], ['A', 'B'])
print(f'Empty LCS: {lcs_empty}')  # Should be []
"
```

Expected: Correct LCS results for known inputs, graceful handling of empty sequences.

Milestone 3 - Diff Generation:

```bash
# Generate actual diff output
python src/main.py test1.txt test2.txt

# Verify unified format structure
python src/main.py test1.txt test2.txt | head -10
```

Expected: Proper unified diff headers, hunk markers with @@ format, correct line prefixes.

Milestone 4 - CLI Interface:

```bash
# Test all CLI options
python src/main.py --help
python src/main.py --context 1 test1.txt test2.txt
python src/main.py --no-color test1.txt test2.txt

# Verify exit codes
python src/main.py test1.txt test1.txt; echo "Exit code: $?"  # Should be 0
python src/main.py test1.txt test2.txt; echo "Exit code: $?"  # Should be 1
```

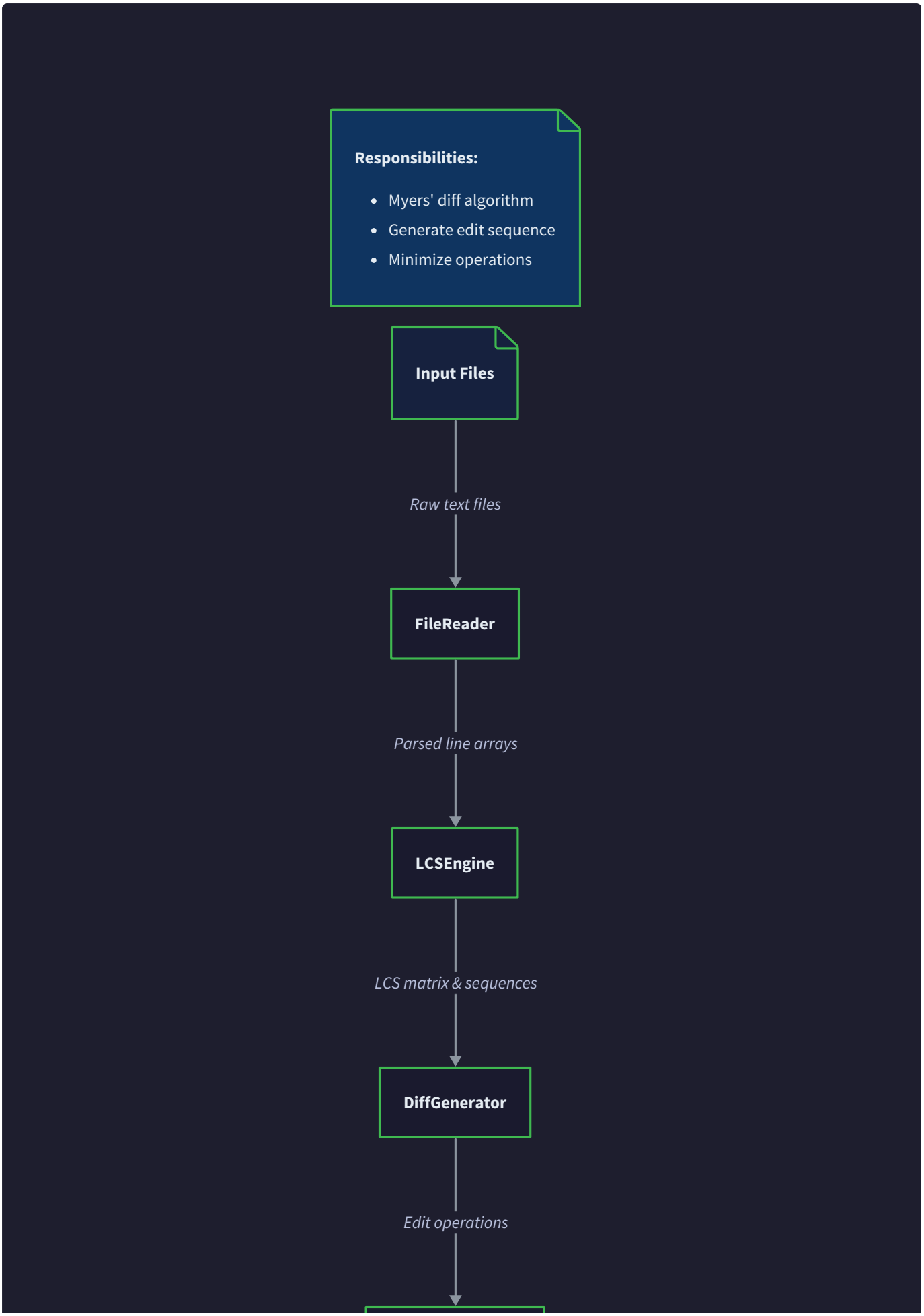
Expected: Help output displays, context option changes output, colors work in terminal but not in pipes, correct exit codes.

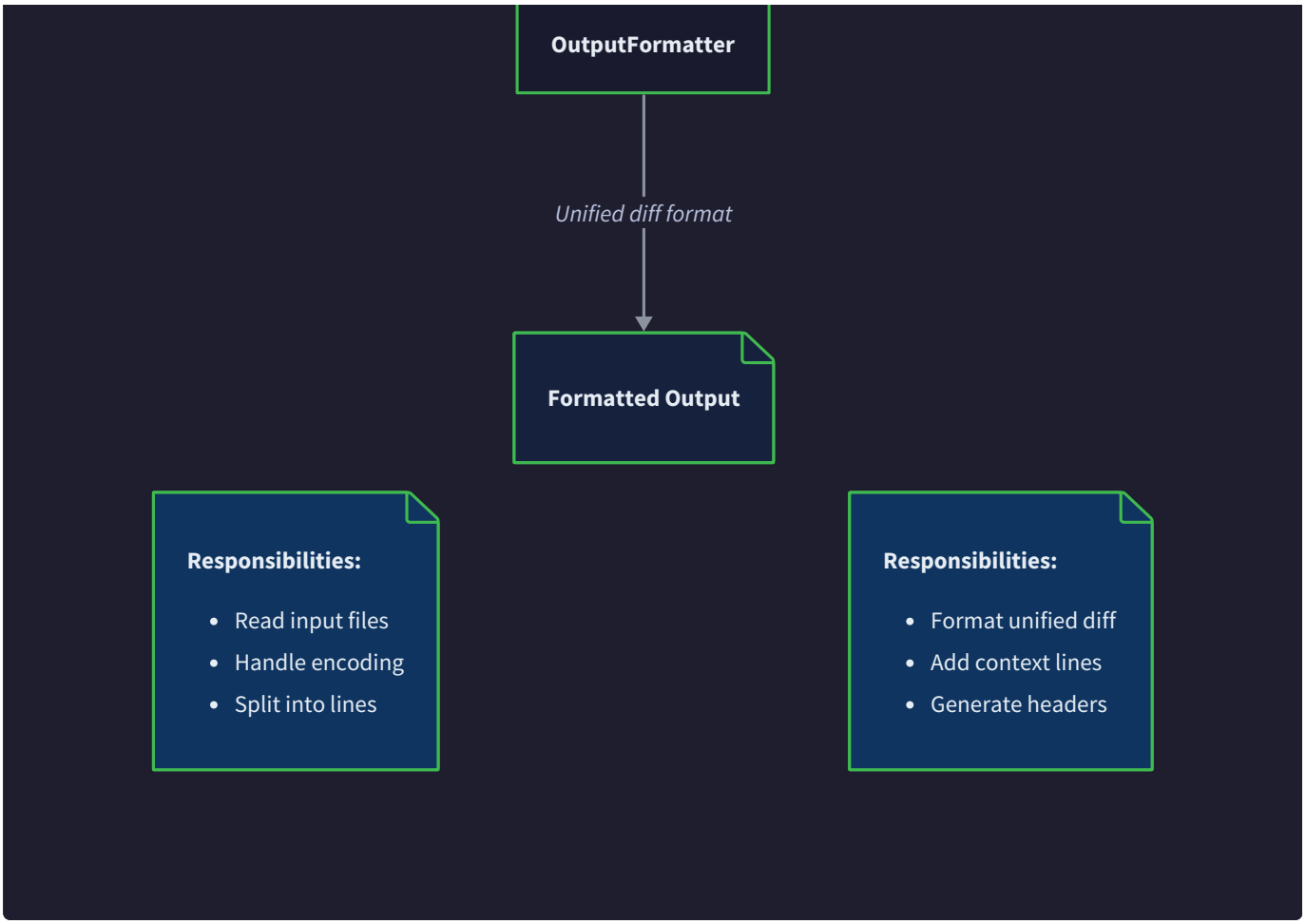
High-Level Architecture

> **Milestone(s):** All milestones — the component architecture forms the foundation for implementing line tokenization, LCS computation, diff generation, and CLI output

The diff tool follows a pipeline architecture where data flows through four specialized components, each with a distinct responsibility. Think of this like an assembly line in a publishing house: manuscripts arrive at the **Document Preparation** station (FileReader) where they're standardized and broken into comparable units, then move to the **Analysis** station (LCSEngine) where editors identify common content, followed by the **Editorial** station (DiffGenerator) where change instructions are created, and finally the **Publishing** station (OutputFormatter) where changes are formatted for readers.

This pipeline design provides clear separation of concerns, making each component independently testable and allowing us to swap implementations (for example, replacing the LCS algorithm with Myers' algorithm) without affecting other parts of the system. The unidirectional data flow also makes the system easier to reason about and debug — we can examine the output of each stage to isolate problems.


Responsibilities:
Myers' diff algorithm
Generate edit sequence
Minimize operations
Input Files
Raw text files
FileReader
Parsed line arrays
LCSEngine
LCS matrix & sequences
DiffGenerator
Edit operations



Component Overview

The diff tool architecture consists of four main components that transform input files into formatted diff output through a series of well-defined stages. Each component has a specific responsibility and interfaces with adjacent components through structured data types.

FileReader Component

The **FileReader** serves as the entry point to our diff pipeline, responsible for converting raw file data into normalized line sequences that can be compared. This component handles the complexities of file encoding detection, line ending normalization, and memory-efficient reading of potentially large files.

Responsibility	Description	Input	Output
Encoding Detection	Automatically detect file encoding (UTF-8, Latin-1)	File path	Encoding identifier
Line Tokenization	Split file content into individual lines	Raw file bytes	Array of line strings
Normalization	Handle different line endings (LF, CRLF, CR)	Raw lines	Normalized line array
Error Handling	Detect binary files and encoding failures	File system data	Error messages or success

The FileReader abstracts away platform-specific file handling concerns, ensuring that subsequent components receive clean, predictable input regardless of the original file format. This isolation is crucial because text files can vary dramatically in encoding, line endings, and structure across different operating systems and editors.

> **Decision: Line-Based Tokenization**
>
> - **Context**: We need to decide the granularity of comparison — character-level, word-level, or line-level tokenization
> - **Options Considered**: Character diff (fine-grained but noisy), word diff (semantic but complex), line diff (standard and readable)
> - **Decision**: Line-based tokenization with optional future word-level support
> - **Rationale**: Line-based diffs match user expectations from tools like `git diff` and `unified diff`, provide good balance between granularity and readability, and align with how developers think about code changes
> - **Consequences**: May miss fine-grained changes within lines, but produces familiar output format that integrates well with existing tooling

LCSEngine Component

The **LCSEngine** implements the core algorithmic logic for finding the longest common subsequence between two line arrays using dynamic programming. This component encapsulates the computational complexity of sequence alignment, providing a clean interface that hides the matrix operations and backtracking logic.

Method	Parameters	Returns	Description
`compute_lcs`	`sequence1: list, sequence2: list`	`CommonSubsequence`	Finds longest common subsequence using DP
`build_matrix`	`lines1: list, lines2: list`	`Matrix[int]`	Constructs LCS length matrix
`backtrack`	`matrix: Matrix, seq1: list, seq2: list`	`list[str]`	Recovers actual LCS from matrix
`optimize_memory`	`matrix: Matrix`	`Matrix[int]`	Applies space optimization for large inputs

The LCSEngine abstracts the mathematical complexity of dynamic programming, allowing other components to request sequence alignment without understanding matrix operations or backtracking algorithms. This separation enables us to experiment with different optimization strategies (like Hirschberg's algorithm for space efficiency) without affecting the rest of the system.

> **Decision: Dynamic Programming Over Naive Comparison**
>
> - **Context**: Multiple algorithms exist for finding common subsequences with different time/space trade-offs
> - **Options Considered**: Naive O(2^n) recursive approach, memoized recursion, bottom-up dynamic programming
> - **Decision**: Bottom-up dynamic programming with optional space optimization
> - **Rationale**: DP provides guaranteed O(mn) time complexity, iterative approach avoids stack overflow on large inputs, and the matrix structure enables future optimizations like Myers' algorithm
> - **Consequences**: Uses O(mn) space which can be prohibitive for very large files, but provides predictable performance and clear upgrade path to space-optimized variants

DiffGenerator Component

The **DiffGenerator** transforms the abstract common subsequence result into concrete diff operations and organizes them into contextual hunks for human consumption. This component bridges between the mathematical result of sequence alignment and the practical needs of presenting changes in a readable format.

Operation Type	Symbol	Description	Context Requirements
UNCHANGED	` `	Lines present in both files	Forms context around changes
ADDED	`+`	Lines only in second file	Grouped into hunks with context
DELETED	`-`	Lines only in first file	Marked with original line numbers
CONTEXT	` `	Surrounding unchanged lines	Configurable count (default 3)

The DiffGenerator must solve the challenging problem of grouping individual edit operations into meaningful chunks while preserving enough context for users to understand the changes. This involves complex logic for determining hunk boundaries, handling overlapping context, and generating accurate line number ranges.

> The critical insight here is that raw edit operations are too granular for human consumption — users need changes grouped into logical chunks with surrounding context to understand the intent behind modifications.

OutputFormatter Component

The **OutputFormatter** handles the final presentation layer, converting structured diff hunks into various output formats with optional visual enhancements. This component manages the complexity of terminal capabilities, color support detection, and format compatibility with standard tools.

Format Type	Use Case	Features	Compatibility
Unified Diff	Standard text output	Header lines, hunk markers, line prefixes	`patch`, `git apply`
Colored Terminal	Interactive viewing	ANSI color codes, TTY detection	Modern terminals
Plain Text	Scripting, pipes	No color codes, clean parsing	All environments
Side-by-Side	Visual comparison	Parallel columns, alignment	Wide terminals

The OutputFormatter must handle the subtleties of terminal capabilities, ensuring that color codes appear only when appropriate and that output remains parseable by other tools when redirected to files or pipes.

Component Communication Patterns

The components communicate through well-defined interfaces that pass structured data types rather than exposing internal implementation details. This design enables testing each component in isolation and supports future enhancements without cascading changes.

Interface	Data Type	Direction	Purpose
FileReader → LCSEngine	`Sequence[str]`	Forward	Normalized line arrays
LCSEngine → DiffGenerator	`CommonSubsequence`	Forward	Alignment result
DiffGenerator → OutputFormatter	`list[Hunk]`	Forward	Structured diff hunks
All Components → CLI	`Result[T, Error]`	Return	Success/failure reporting

The unidirectional flow simplifies reasoning about data transformations and makes the system more predictable. Each component can focus on its specific responsibility without worrying about side effects or circular dependencies.

Recommended File Structure

The codebase organization reflects the component architecture, with each major component isolated in its own module to support independent development and testing. This structure helps prevent circular dependencies and makes the system easier to understand for new contributors.

```
diff-tool/
├── src/
│   ├── __init__.py                ← Package initialization
│   ├── main.py                    ← CLI entry point and argument parsing
│   ├── file_reader.py             ← FileReader component implementation
│   ├── lcs_engine.py              ← LCSEngine with dynamic programming
│   ├── diff_generator.py          ← DiffGenerator for hunk creation
│   ├── output_formatter.py        ← OutputFormatter with color support
│   └── types.py                   ← Shared data structures and enums
├── tests/
│   ├── test_file_reader.py        ← FileReader component tests
│   ├── test_lcs_engine.py         ← LCS algorithm correctness tests
│   ├── test_diff_generator.py     ← Diff generation and hunk tests
│   ├── test_output_formatter.py   ← Format and color output tests
│   ├── test_integration.py        ← End-to-end pipeline tests
│   └── fixtures/                  ← Test files with various encodings
│       ├── utf8_sample.txt
│       ├── latin1_sample.txt
│       ├── mixed_endings.txt
│       └── binary_file.bin
└── README.md                      ← Usage examples and installation
```

This structure separates concerns at the file level, making it easy to locate and modify specific functionality. The `types.py` module contains shared data structures, preventing circular imports while keeping related types together.

> **Decision: Separate Module per Component**
>
> - **Context**: We need to organize code to support independent development and testing of each component
> - **Options Considered**: Single large file, functional modules, class-based components in separate files
> - **Decision**: One module per major component with shared types module
> - **Rationale**: Enables parallel development, simplifies testing setup, prevents circular dependencies, and makes the codebase easier to navigate for newcomers
> - **Consequences**: More files to manage but clearer separation of responsibilities and better support for future team development

The test structure mirrors the source structure, with additional integration tests to verify the complete pipeline. The fixtures directory provides a variety of test files to validate handling of different encodings and edge cases.

Data Flow Architecture

The pipeline processes data through distinct transformation stages, with each component adding structure and semantic meaning to the information. Understanding this flow is crucial for debugging issues and optimizing performance.

Stage 1: File Input → Line Sequences Raw file bytes are transformed into normalized arrays of strings, with encoding detection and line ending standardization applied. The FileReader produces clean, comparable

sequences regardless of input file variations.

Stage 2: Line Sequences → Common Subsequence
The LCSEngine processes both line arrays simultaneously, building a dynamic programming matrix to identify the optimal alignment. The result identifies which lines are shared between the files and in what order.

Stage 3: Common Subsequence → Edit Operations The DiffGenerator compares the original sequences against the common subsequence to determine which lines were added, deleted, or unchanged. These operations are then grouped into hunks with appropriate context.

Stage 4: Edit Operations → Formatted Output The OutputFormatter converts the structured hunks into human-readable text, applying color formatting when appropriate and generating output compatible with standard diff tools.

Stage	Input Type	Processing	Output Type	Key Challenge
1	Raw bytes	Encoding + tokenization	`Sequence[str]`	Encoding detection
2	Two sequences	Dynamic programming	`CommonSubsequence`	Memory efficiency
3	Sequences + LCS	Edit script generation	`list[Hunk]`	Context grouping
4	Structured hunks	Format generation	Formatted text	Color/compatibility

This staged approach allows us to validate data at each transformation point and provides clear interfaces for testing. Each stage adds semantic value while maintaining the information needed for subsequent processing.

Common Pitfalls

⚠️ **Pitfall: Circular Dependencies Between Components** Component modules importing each other creates circular dependency errors and makes testing difficult. This typically happens when components directly reference each other's implementation details rather than using shared data types. Fix by moving shared types to a separate module and ensuring unidirectional data flow.

⚠️ **Pitfall: Tight Coupling Through Implementation Details** Components that depend on internal implementation details of other components become fragile and hard to modify. For example, if DiffGenerator directly accesses LCSEngine's matrix rather than using the public interface, changes to the matrix representation break the diff generator. Always use well-defined interfaces and data types.

⚠️ **Pitfall: Missing Error Propagation Between Components** Errors that occur in one component but aren't properly propagated to the CLI result in confusing failure modes where the program appears to succeed but produces incorrect output. Each component must properly handle errors from upstream components and provide meaningful error messages.

⚠️ **Pitfall: Memory Accumulation Across Pipeline Stages** Keeping references to large intermediate data structures throughout the pipeline can cause memory usage to grow unnecessarily. For example, retaining the full LCS matrix after backtracking completes wastes space. Design components to release large intermediate data once no longer needed.

Implementation Guidance

The component architecture provides a clean foundation for implementation, with each component focused on a specific aspect of the diff computation. This guidance helps translate the architectural design into working code.

Technology Recommendations

Component	Simple Approach	Advanced Approach	Recommendation
File Reading	`open()` with try/except	`chardet` library + streaming	Simple for learning
LCS Computation	2D list matrix	NumPy arrays	Simple 2D list
Hunk Generation	List comprehensions	Generator functions	List comprehensions
Color Output	String concatenation	`colorama` library	String concatenation
CLI Parsing	`sys.argv` manually	`argparse` module	`argparse` module

For a learning project, the simple approaches provide better understanding of the underlying concepts without external dependencies obscuring the core algorithms.

Project Structure Setup

Start by creating the directory structure and basic module files:

PYTHON

```python
# src/types.py

"""Shared data structures for the diff tool pipeline."""

from dataclasses import dataclass

from typing import List, Optional

from enum import Enum

class LineType(Enum):

    UNCHANGED = "unchanged"

    ADDED = "added"

    DELETED = "deleted"

@dataclass

class DiffLine:

    content: str

    line_type: LineType

    old_line_num: Optional[int] = None

    new_line_num: Optional[int] = None

@dataclass

class Hunk:

    old_start: int

    old_count: int

    new_start: int

    new_count: int

    lines: List[DiffLine]

@dataclass

class CommonSubsequence:

    elements: List[str]
```

```
    positions_seq1: List[int]

    positions_seq2: List[int]
```

Core Component Interfaces

Each component should expose a clean interface that hides implementation complexity:

PYTHON

```python
# src/file_reader.py

"""File reading and line tokenization component."""


def detect_file_encoding(filepath: str) -> str:

    """Determine file encoding, trying UTF-8 first, then Latin-1.



    Args:

        filepath: Path to the file to analyze



    Returns:

        Encoding name ('utf-8' or 'latin-1')



    Raises:

        FileNotFoundError: If file doesn't exist

        PermissionError: If file can't be read

    """

    # TODO 1: Try to open file with UTF-8 encoding

    # TODO 2: If UnicodeDecodeError, try Latin-1

    # TODO 3: If both fail, raise encoding detection error

    # TODO 4: Return successful encoding name

    pass


def read_file_lines(filepath: str) -> List[str]:

    """Read file with encoding detection and line normalization.



    Args:

        filepath: Path to the file to read



    Returns:
```

```
        List of lines with normalized endings (no trailing \\n)


    Raises:
        FileNotFoundError: If file doesn't exist
        PermissionError: If file can't be read
        UnicodeDecodeError: If file encoding cannot be determined
    """
    # TODO 1: Detect file encoding using detect_file_encoding()
    # TODO 2: Open file with detected encoding
    # TODO 3: Read all lines preserving empty lines
    # TODO 4: Normalize line endings (remove \\r\\n, \\r, \\n)
    # TODO 5: Return normalized line list
    pass
```

PYTHON

```python
# src/lcs_engine.py

"""Longest Common Subsequence computation using dynamic programming."""

from typing import List

from .types import CommonSubsequence

def compute_lcs(sequence1: List[str], sequence2: List[str]) -> CommonSubsequence:

    """Find longest common subsequence using dynamic programming.



    Args:

        sequence1: First sequence (lines from file 1)

        sequence2: Second sequence (lines from file 2)



    Returns:

        CommonSubsequence containing the LCS and position mappings

    """

    # TODO 1: Handle empty sequence edge cases

    # TODO 2: Build LCS length matrix using dynamic programming

    # TODO 3: Backtrack through matrix to find actual LCS elements

    # TODO 4: Record positions in both original sequences

    # TODO 5: Return CommonSubsequence with elements and positions

    pass

def build_lcs_matrix(seq1: List[str], seq2: List[str]) -> List[List[int]]:

    """Build the LCS length matrix using dynamic programming.



    Args:

        seq1: First sequence

        seq2: Second sequence
```

```
    Returns:
        2D matrix where matrix[i][j] = LCS length of seq1[:i] and seq2[:j]
    """
    # TODO 1: Initialize matrix with dimensions (len(seq1)+1) x (len(seq2)+1)
    # TODO 2: Fill first row and column with zeros
    # TODO 3: For each cell, if elements match: matrix[i][j] = matrix[i-1][j-1] + 1
    # TODO 4: If elements don't match: matrix[i][j] = max(matrix[i-1][j], matrix[i][j-1])
    # TODO 5: Return completed matrix
    pass
```

Component Integration Pattern

Use a main pipeline function that coordinates the components:

PYTHON

```python
# src/main.py

"""Main CLI entry point and component coordination."""


import sys

import argparse

from typing import Optional


from .file_reader import read_file_lines

from .lcs_engine import compute_lcs

from .diff_generator import generate_diff

from .output_formatter import format_unified_diff, ColorFormatter


def main():

    """Main entry point for diff tool CLI."""

    # TODO 1: Parse command line arguments (file1, file2, --context, --no-color)

    # TODO 2: Read both files using FileReader component

    # TODO 3: Compute LCS using LCSEngine component

    # TODO 4: Generate diff hunks using DiffGenerator component

    # TODO 5: Format and output using OutputFormatter component

    # TODO 6: Set appropriate exit code (0 if same, 1 if different)

    pass


def create_argument_parser() -> argparse.ArgumentParser:

    """Create CLI argument parser with diff tool options."""

    # TODO 1: Create ArgumentParser with description

    # TODO 2: Add positional arguments for file1 and file2

    # TODO 3: Add --context option with integer argument (default 3)

    # TODO 4: Add --no-color flag for plain text output

    # TODO 5: Return configured parser

    pass
```

```
if __name__ == "__main__":
    main()
```

Milestone Checkpoints

After Milestone 1 (Line Tokenization): Run `python -m src.file_reader test_file.txt` to verify file reading works correctly. You should see line count output and proper handling of different encodings. Test with UTF-8, Latin-1, and binary files to ensure encoding detection works.

After Milestone 2 (LCS Algorithm):
Run `python -c "from src.lcs_engine import compute_lcs; print(compute_lcs(['a','b','c'], ['a','c','d']))"` to verify LCS computation. The result should show common subsequence ['a','c'] with correct position mappings.

After Milestone 3 (Diff Generation): Run `python -m src.main file1.txt file2.txt` to see unified diff output. Verify that hunks are properly formed with @@ headers and correct line numbers.

After Milestone 4 (CLI and Color): Test `python -m src.main file1.txt file2.txt --context 5 --no-color` to verify all CLI options work correctly. Check that colors appear in terminal but not when redirected to a file.

Language-Specific Implementation Notes

Python File Handling:

- Use `open(filepath, 'r', encoding=encoding)` for text files
- Handle `UnicodeDecodeError` when trying different encodings
- Use `str.splitlines(keepends=False)` to normalize line endings
- Check `sys.stdout.isatty()` for TTY detection

Python Data Structures:

- Use `List[List[int]]` for the LCS matrix (simple and clear)
- Use `dataclasses` for structured types like `Hunk` and `DiffLine`
- Use `typing.Optional` for fields that may be None
- Use `enum.Enum` for constants like line types and colors

Python Testing:

- Use `unittest` module for component tests
- Create fixtures in `tests/fixtures/` directory
- Use `tempfile` module for creating test files
- Mock file system errors using `unittest.mock`

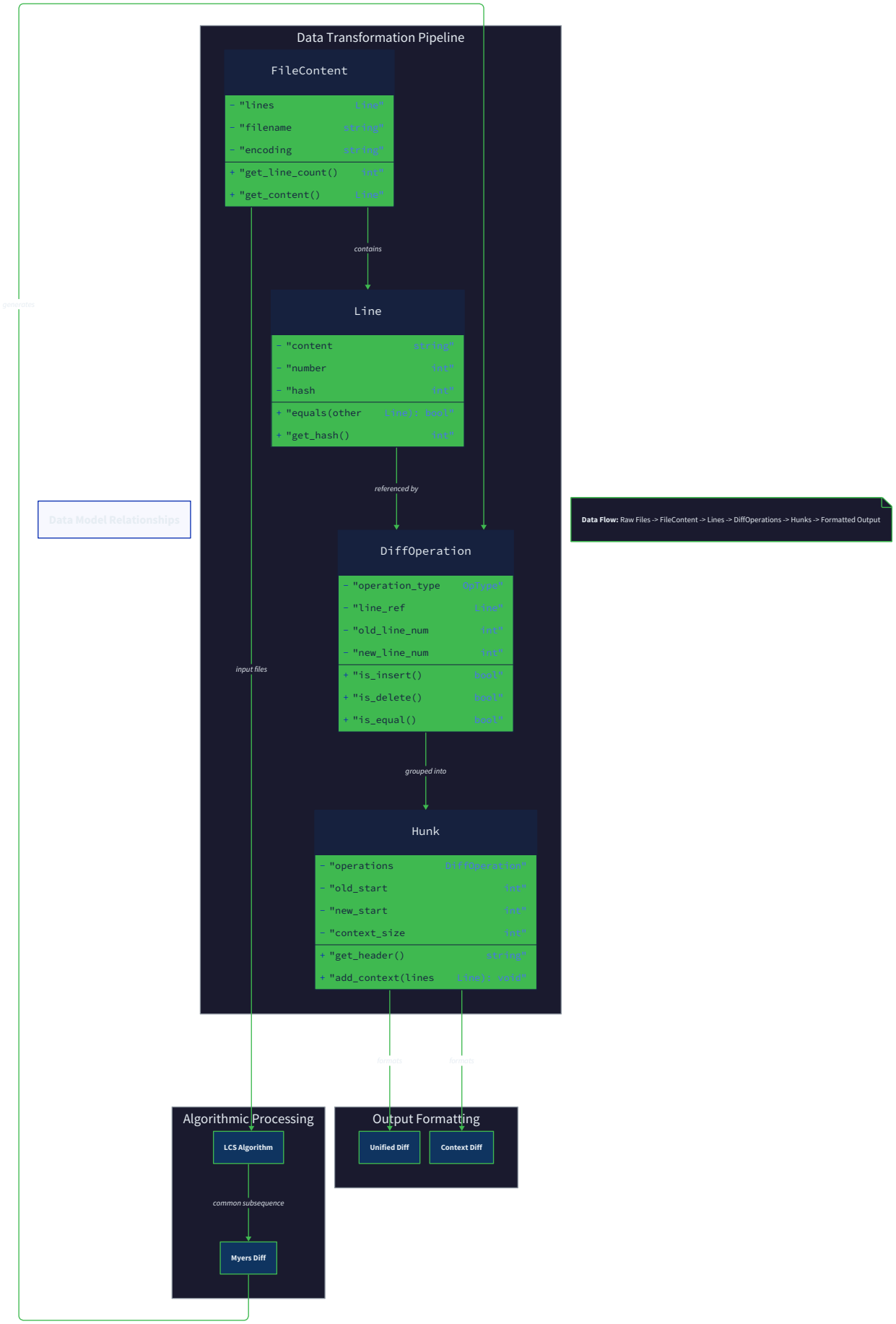
This implementation guidance provides the scaffolding for building the diff tool while leaving the core algorithmic challenges for the learner to solve. The clear separation of components makes testing straightforward and debugging manageable.

Data Model

> **Milestone(s):** All milestones — the data model forms the foundation that connects line tokenization (Milestone 1), LCS computation (Milestone 2), diff generation (Milestone 3), and CLI output (Milestone 4)

The data model serves as the architectural backbone of our diff tool, defining how information flows from raw file content through algorithmic processing to formatted output. Think of the data model as the **vocabulary and grammar** of our diff system — just as human language needs nouns, verbs, and sentence structures to convey meaning, our diff tool needs well-defined types and relationships to represent files, changes, and output formats.

Understanding the data model is crucial because it determines how each component communicates with others. A poorly designed data model leads to awkward conversions, performance bottlenecks, and maintenance headaches. Our design prioritizes clarity and efficiency, ensuring that data transformations feel natural and that the types directly reflect the problem domain.


Data Transformation Pipeline
FileContent
- "lines Line"
- "filename string"
- "encoding string"
+ "get_line_count() int"
+ "get_content() Line"
contains
Line
- "content string"
- "number int"
- "hash int"
+ "equals(other Line): bool"
+ "get_hash() int"
referenced by
DiffOperation
- "operation_type OpType"
- "line_ref Line"
- "old_line_num int"
- "new_line_num int"
+ "is_insert() bool"
+ "is_delete() bool"
+ "is_equal() bool"
grouped into
Hunk
- "operations DiffOperation"
- "old_start int"
- "new_start int"
- "context_size int"
+ "get_header() string"
+ "add_context(lines Line): void"
generates
input files
formats
formats
Data Model Relationships
Data Flow: Raw Files -> FileContent -> Lines -> DiffOperations -> Hunks -> Formatted Output
Algorithmic Processing
LCS Algorithm
common subsequence
Myers Diff
Output Formatting
Unified Diff
Context Diff

The mental model for our data structures follows the **document comparison workflow**: we start with raw files, break them into comparable units (lines), find commonalities and differences, group related changes together, and finally format them for human consumption. Each data structure represents a specific stage in this transformation pipeline.

Core Types and Structures

Our data model consists of four primary structures that represent different abstraction levels in the diff process. Each structure encapsulates specific responsibilities and provides a clean interface for the components that manipulate it.

FileContent Structure

The `FileContent` structure represents the parsed and normalized content of a single input file. This is the first structured representation we create from raw file data, serving as the foundation for all subsequent processing.

Field Name	Type	Description
`filepath`	`str`	Original file path for error reporting and output headers
`lines`	`list[str]`	Normalized lines with consistent line endings removed
`line_count`	`int`	Total number of lines including empty lines
`encoding`	`str`	Detected encoding used to read the file (UTF-8, Latin-1)
`original_endings`	`str`	Original line ending type detected (LF, CRLF, CR) for preservation

The `FileContent` structure encapsulates the preprocessing decisions made during file reading. By storing the original encoding and line endings, we preserve information that might be needed for error reporting or round-trip compatibility. The normalized `lines` array provides a clean interface for comparison algorithms while maintaining traceability to the original file structure.

> **Design Insight**: Separating file metadata from content allows components to work with clean line arrays while preserving the ability to generate accurate error messages and output headers that reference original file characteristics.

DiffLine Structure

The `DiffLine` structure represents a single line in the final diff output, combining content with presentation metadata. This structure bridges the gap between algorithmic results and formatted output.

Field Name	Type	Description
`content`	`str`	The actual line text without diff prefixes or formatting
`line_type`	`LineType`	Classification as UNCHANGED, ADDED, or DELETED
`old_line_num`	`Optional[int]`	Line number in original file (None for ADDED lines)
`new_line_num`	`Optional[int]`	Line number in modified file (None for DELETED lines)

The dual line numbering system enables generation of accurate unified diff headers while supporting other output formats. The optional nature of line numbers reflects the logical reality that added lines don't exist in the original file and deleted lines don't exist in the modified file.

LineType Enumeration

The `LineType` enumeration provides a clear classification system for diff operations, avoiding magic strings or numeric codes that could lead to confusion.

Enum Value	Description	Unified Diff Prefix
`UNCHANGED`	Line exists in both files with identical content	` ` (space)
`ADDED`	Line exists only in the new file	`+`
`DELETED`	Line exists only in the old file	`-`

> **Decision: Explicit LineType Enumeration**
>
> - **Context**: Need to classify lines for diff output formatting
> - **Options Considered**: String constants ("added", "deleted"), integer codes (0, 1, 2), enumeration
> - **Decision**: Use explicit enumeration with descriptive names
> - **Rationale**: Type safety prevents invalid values, IDE autocompletion reduces errors, self-documenting code
> - **Consequences**: Slightly more verbose than strings but eliminates entire class of typo bugs

Hunk Structure

The `Hunk` structure represents a group of consecutive changes along with surrounding context lines. This is the fundamental unit of diff output, corresponding to the sections marked with `@@` headers in unified diff format.

Field Name	Type	Description
`old_start`	`int`	Starting line number in original file (1-indexed for diff format)
`old_count`	`int`	Number of lines from original file included in this hunk
`new_start`	`int`	Starting line number in modified file (1-indexed for diff format)
`new_count`	`int`	Number of lines from modified file included in this hunk
`lines`	`list[DiffLine]`	All lines in this hunk including context and changes
`context_before`	`int`	Number of context lines before first change
`context_after`	`int`	Number of context lines after last change

The hunk structure encapsulates both the logical grouping of changes and the metadata needed to generate proper unified diff headers. The context tracking enables intelligent hunk merging when changes are close together.

> **Design Insight**: Storing context line counts separately from the line array enables algorithms to distinguish between context and actual changes without scanning the entire line list.

CommonSubsequence Structure

The `CommonSubsequence` structure represents the result of LCS computation, containing both the matching elements and metadata about the comparison process.

Field Name	Type	Description
`elements`	`list[str]`	The actual longest common subsequence of lines
`length`	`int`	Length of the common subsequence for quick access
`positions1`	`list[int]`	Indices of LCS elements in the first sequence
`positions2`	`list[int]`	Indices of LCS elements in the second sequence

The position arrays enable efficient conversion from LCS results to edit operations without re-scanning the input sequences. This design trades memory for computational efficiency, a worthwhile exchange given that diff operations are typically performed once per file pair.

EditDistance Structure

The `EditDistance` structure captures metrics about the differences between two files, useful for summary reporting and algorithm validation.

Field Name	Type	Description
`insertions`	`int`	Number of lines added in the new file
`deletions`	`int`	Number of lines removed from the old file
`unchanged`	`int`	Number of lines that remain identical
`total_operations`	`int`	Sum of insertions and deletions (Levenshtein distance)

This structure provides quick access to diff statistics without requiring traversal of the entire diff output. It supports both user-facing summary information and internal algorithm validation.

Type Relationships

The data structures form a **transformation pipeline** where each type represents a different level of abstraction and processing. Understanding these relationships is crucial for implementing the component interfaces correctly.

File Processing Flow

The transformation begins with raw file data and progresses through increasingly structured representations:

1. **Raw File Data** → `FileContent`: The file reader component processes binary file content, detects encoding, normalizes line endings, and splits into line arrays. This transformation handles the messiness of real-world files and produces clean input for algorithms.
2. **FileContent Pairs** → `CommonSubsequence`: The LCS engine takes two `FileContent` structures and produces a `CommonSubsequence` representing the optimal alignment. This is where the core algorithmic work happens, using dynamic programming to find the longest matching sequence.
3. **CommonSubsequence** + **FileContent Pairs** → **Edit Operations**: The diff generator combines the LCS result with the original file contents to produce a sequence of edit operations. This step determines which lines are insertions, deletions, or unchanged content.
4. **Edit Operations** → `Hunk` **List**: The diff generator groups consecutive changes into hunks, adding configurable context lines around each change group. This transformation makes the diff human-readable by providing surrounding context.
5. **Hunk List** → **Formatted Output**: The output formatter takes the structured hunk representation and generates the final diff format, whether unified diff, colored terminal output, or other presentation formats.

Data Flow Dependencies

The type relationships create specific dependency requirements that influence component design:

Source Type	Target Type	Transformation Component	Key Operations
Raw bytes	`FileContent`	FileReader	Encoding detection, line splitting, normalization
`FileContent` × 2	`CommonSubsequence`	LCSEngine	Dynamic programming matrix computation
`CommonSubsequence` + `FileContent` × 2	`list[DiffLine]`	DiffGenerator	Edit script generation, line classification
`list[DiffLine]`	`list[Hunk]`	DiffGenerator	Context addition, hunk boundary detection
`list[Hunk]`	Formatted output	OutputFormatter	Unified diff formatting, color application

Memory and Performance Implications

The type relationships have important implications for memory usage and performance characteristics:

Memory Growth Pattern: Each transformation step typically increases memory usage as we add metadata and structure to the raw content. The peak memory usage occurs during hunk generation when we hold both the original file contents and the complete diff structure simultaneously.

Processing Efficiency: The position arrays in `CommonSubsequence` eliminate the need for repeated searches during edit script generation. Similarly, storing line numbers in `DiffLine` avoids recalculation during output formatting.

Streaming Opportunities: While the current design loads entire files into memory, the type structure supports future streaming implementations. The `Hunk` structure could be generated and output incrementally for large files.

Error Propagation Through Types

Each type transformation introduces potential failure modes that must be handled gracefully:

Transformation	Potential Failures	Error Information Preserved
Bytes → `FileContent`	Encoding errors, I/O failures	Original filepath, detected encoding
`FileContent` → `CommonSubsequence`	Memory exhaustion	File sizes, available memory
`CommonSubsequence` → Edit operations	Logic errors in backtracking	LCS length, sequence lengths
Edit operations → `Hunk`	Configuration errors	Context line settings, hunk boundaries

The data model preserves enough context information to generate meaningful error messages that help users diagnose problems with their input files or configuration.

Architecture Decision Records

Decision: Separate Line Content from Metadata

- **Context**: Need to represent lines in diff output with various metadata (line numbers, change type, formatting)
- **Options Considered**: Store everything in strings with prefixes, use tuples, create structured DiffLine type
- **Decision**: Create structured `DiffLine` type with separate content and metadata fields
- **Rationale**: Enables clean separation between algorithmic processing (works on content) and presentation (uses metadata), supports multiple output formats without re-parsing
- **Consequences**: Slightly higher memory usage but eliminates string parsing and enables type-safe metadata access

Decision: Position Arrays in CommonSubsequence

- **Context**: Need to convert LCS results into edit operations efficiently
- **Options Considered**: Store only the LCS elements, include position information, use iterator-based approach
- **Decision**: Include position arrays mapping LCS elements back to original sequences
- **Rationale**: Eliminates $O(n^2)$ search during edit script generation, enables parallel processing of multiple LCS results
- **Consequences**: Higher memory usage but significantly faster diff generation for large files

Decision: Context Metadata in Hunk Structure

- **Context**: Need to distinguish between actual changes and context lines within hunks
- **Options Considered**: Mark context lines in DiffLine type, store context counts in Hunk, compute context dynamically
- **Decision**: Store context counts as separate fields in Hunk structure
- **Rationale**: Enables hunk merging algorithms without scanning line arrays, supports configurable context without re-processing
- **Consequences**: Small amount of redundant information but significantly cleaner hunk manipulation code

Common Pitfalls

⚠️ **Pitfall: Mixing Zero-Based and One-Based Indexing** The internal algorithms use zero-based indexing for array access, but unified diff format requires one-based line numbers. Mixing these conventions leads to off-by-one errors in output. Always convert to one-based indexing only in the final formatting stage, and document which fields use which convention.

⚠️ **Pitfall: Forgetting Optional Line Numbers** `DiffLine` uses optional line numbers because added lines don't exist in the original file and deleted lines don't exist in the new file. Code that assumes line numbers are always present will crash on these cases. Always check for `None` before using line numbers or provide sensible defaults for display.

⚠️ **Pitfall: Immutable vs Mutable Structures** The data structures are designed to be immutable after creation to prevent accidental modification during processing. Code that tries to modify structures in-place (like changing `line_type` after creation) violates this design and can lead to inconsistent state. Create new instances rather than modifying existing ones.

⚠️ **Pitfall: Context Line Double-Counting** When generating hunks, it's easy to double-count context lines that appear in multiple hunks or include context lines in the change counts. The `old_count` and `new_count` fields should include context lines, while the separate context counters track how many lines are context versus actual changes.

⚠️ **Pitfall: Encoding Information Loss** The `FileContent` structure preserves original encoding information, but it's easy to lose this during processing and end up with encoding errors in output. Always propagate encoding information through the pipeline and use it when generating error messages or file headers.

Implementation Guidance

The data model implementation focuses on creating clean, type-safe structures that guide correct usage and prevent common mistakes. The design emphasizes immutability and clear ownership to support both single-threaded and potential future multi-threaded implementations.

Technology Recommendations

Component	Simple Option	Advanced Option
Type Definitions	`dataclass` with frozen=True	`pydantic` models with validation
Enumerations	`enum.Enum` with string values	`enum.Enum` with custom methods
Optional Types	`typing.Optional` for clarity	`typing.Union` with None
Collections	Built-in `list` and `dict`	`typing.NamedTuple` for small structures

Recommended File Structure

```
diff_tool/
  models/
    __init__.py              ← export all public types
    file_content.py          ← FileContent and related types
    diff_line.py             ← DiffLine and LineType enum
    hunk.py                  ← Hunk structure and hunk-related logic
    lcs_types.py             ← CommonSubsequence and EditDistance
  tests/
    test_models/
      test_file_content.py   ← test file content creation and validation
      test_diff_line.py      ← test line type behavior and formatting
      test_hunk.py           ← test hunk creation and manipulation
```

Core Type Definitions

```python
from dataclasses import dataclass

from enum import Enum

from typing import List, Optional


@dataclass(frozen=True)

class FileContent:

    """Represents processed file content ready for comparison."""

    filepath: str

    lines: List[str]

    line_count: int

    encoding: str

    original_endings: str



    def __post_init__(self):

        # TODO: Validate that line_count matches len(lines)

        # TODO: Validate that encoding is supported ('utf-8', 'latin-1')

        # TODO: Validate that original_endings is valid ('LF', 'CRLF', 'CR')

        pass


class LineType(Enum):

    """Classification of lines in diff output."""

    UNCHANGED = "unchanged"

    ADDED = "added"

    DELETED = "deleted"



    def to_diff_prefix(self) -> str:

        """Convert to unified diff prefix character."""

        # TODO: Return ' ' for UNCHANGED
```

```
        # TODO: Return '+' for ADDED
        # TODO: Return '-' for DELETED
        pass

@dataclass(frozen=True)
class DiffLine:
    """Single line in diff output with metadata."""
    content: str
    line_type: LineType
    old_line_num: Optional[int]
    new_line_num: Optional[int]

    def __post_init__(self):
        # TODO: Validate that ADDED lines have old_line_num = None
        # TODO: Validate that DELETED lines have new_line_num = None
        # TODO: Validate that UNCHANGED lines have both line numbers
        pass

@dataclass(frozen=True)
class CommonSubsequence:
    """Result of LCS computation with position tracking."""
    elements: List[str]
    length: int
    positions1: List[int]
    positions2: List[int]

    def __post_init__(self):
        # TODO: Validate that length == len(elements)
        # TODO: Validate that len(positions1) == len(positions2) == length
```

```
        # TODO: Validate that positions are in ascending order
        pass

@dataclass(frozen=True)
class Hunk:
    """Group of consecutive changes with context."""
    old_start: int
    old_count: int
    new_start: int
    new_count: int
    lines: List[DiffLine]
    context_before: int
    context_after: int

    def format_header(self) -> str:
        """Generate unified diff hunk header."""
        # TODO: Format as "@@ -old_start,old_count +new_start,new_count @@"
        # TODO: Handle special case where count is 1 (omit count in header)
        # TODO: Ensure line numbers are 1-indexed for output
        pass

@dataclass(frozen=True)
class EditDistance:
    """Metrics about differences between files."""
    insertions: int
    deletions: int
    unchanged: int
    total_operations: int
```

```
    @classmethod
    def from_diff_lines(cls, lines: List[DiffLine]) -> 'EditDistance':
        """Compute edit distance metrics from diff output."""
        # TODO: Count lines by LineType
        # TODO: Calculate total_operations as insertions + deletions
        # TODO: Validate that counts are non-negative
        pass
```

Helper Functions

```python
def create_file_content(filepath: str, raw_content: bytes, encoding: str,
                        line_ending: str) -> FileContent:
    """Factory function for FileContent creation with validation."""
    # TODO: Decode raw_content using specified encoding
    # TODO: Split content into lines preserving empty lines
    # TODO: Normalize line endings to \n for internal processing
    # TODO: Count total lines including final empty line if present
    # TODO: Return FileContent with all fields populated
    pass

def merge_hunks(hunk1: Hunk, hunk2: Hunk, max_gap: int = 3) -> Optional[Hunk]:
    """Merge two adjacent hunks if they're close enough."""
    # TODO: Check if hunks are adjacent (gap <= max_gap)
    # TODO: Combine line arrays with bridging context
    # TODO: Recalculate hunk boundaries and counts
    # TODO: Return merged hunk or None if unmergeable
    pass

def validate_hunk_consistency(hunk: Hunk) -> bool:
    """Verify that hunk line counts match actual line content."""
    # TODO: Count UNCHANGED and DELETED lines, verify equals old_count
    # TODO: Count UNCHANGED and ADDED lines, verify equals new_count
    # TODO: Check that context counts don't exceed total lines
    # TODO: Verify that line numbers are consecutive within each file
    pass
```

Type Conversion Utilities

```python
def diff_lines_to_hunks(lines: List[DiffLine], context_lines: int = 3) -> List[Hunk]:
    """Convert flat diff line list to structured hunks with context."""
    # TODO: Identify change boundaries (sequences of ADDED/DELETED lines)
    # TODO: Add context_lines before and after each change group
    # TODO: Merge hunks that overlap due to context
    # TODO: Calculate correct line numbers and counts for each hunk
    # TODO: Return list of Hunk objects ready for formatting
    pass

def lcs_to_edit_operations(lcs: CommonSubsequence, file1: FileContent,
                           file2: FileContent) -> List[DiffLine]:
    """Convert LCS result to sequence of edit operations."""
    # TODO: Walk through both files using LCS positions as anchors
    # TODO: Lines in LCS become UNCHANGED DiffLines
    # TODO: Lines in file1 not in LCS become DELETED DiffLines
    # TODO: Lines in file2 not in LCS become ADDED DiffLines
    # TODO: Assign correct line numbers from original files
    pass
```

Milestone Checkpoints

After implementing basic types:

- Run `python -m pytest tests/test_models/` — all type creation and validation tests should pass
- Create sample `FileContent` objects — verify encoding and line count are correct
- Test `LineType` enum — verify `to_diff_prefix()` returns correct characters
- Check immutability — attempting to modify frozen dataclass fields should raise `FrozenInstanceError`

After implementing conversion functions:

- Test `create_file_content()` with files containing different line endings — should normalize internally but preserve original format metadata
- Test `diff_lines_to_hunks()` with sample edit sequences — verify context lines and hunk boundaries are correct

- Verify hunk merging logic — adjacent hunks within context distance should merge, distant hunks should remain separate

After implementing validation:

- Test edge cases: empty files, files with only additions/deletions, files with no common lines
- Verify error handling: invalid encodings, inconsistent line counts, malformed hunk data
- Check memory usage with large files — data structures should scale linearly with file size

Signs of correct implementation:

- Type constructors reject invalid combinations (e.g., ADDED line with old_line_num set)
- Conversion between types preserves information (round-trip testing)
- Hunk generation produces output compatible with standard diff tools
- Memory usage grows predictably with input size

Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
Off-by-one errors in diff output	Mixing zero-based and one-based indexing	Print line numbers at each conversion step	Use zero-based internally, convert to one-based only for output
Missing lines in hunks	Incorrect context line calculation	Check `context_before` and `context_after` values	Ensure context doesn't exceed file boundaries
Hunk merge failures	Overlapping context not detected	Log hunk boundaries before merge attempts	Fix gap calculation to account for context overlap
Type validation errors	Invalid field combinations	Enable dataclass validation in `__post_init__`	Add comprehensive validation for all field constraints
Memory errors with large files	Storing redundant data	Profile memory usage by component	Consider streaming or lazy evaluation for large files

File Reader Component

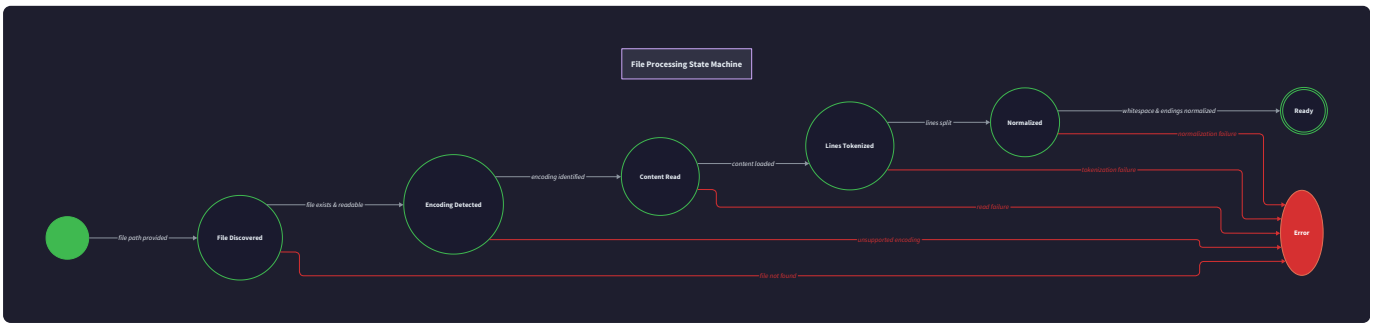
> **Milestone(s):** Milestone 1: Line Tokenization — handles file reading, encoding detection, line splitting, and text normalization that forms the foundation for all subsequent comparison operations

The File Reader Component serves as the entry point to our diff tool, responsible for transforming raw file data into structured, comparable sequences of lines. This component must handle the messy realities of text files while providing clean, normalized input to downstream components.

Mental Model: Document Preparation

Think of the File Reader Component like a librarian preparing two historical manuscripts for scholarly comparison. Before scholars can analyze differences in content, the librarian must first standardize the physical presentation: ensuring both documents use the same character encoding (like translating from different alphabets to a common script), normalizing page breaks and line endings (like converting scrolls and bound books to the same format), and cataloging each line with consistent numbering. Just as manuscripts might use different writing systems, line spacing, or organizational structures, text files arrive with varying encodings, line endings, and formatting conventions that must be reconciled before meaningful comparison can begin.

The librarian's careful preparation work — though invisible to the final scholarly analysis — determines whether the comparison will be accurate and meaningful. Similarly, our File Reader Component performs the crucial but often overlooked work of transforming arbitrary text files into clean, comparable data structures that enable accurate diff computation.



The component operates as a state machine, progressing through distinct phases: initial file discovery, encoding detection, content reading, line tokenization, and normalization. Each state transition represents a validation checkpoint where errors can be detected and handled gracefully.

Encoding Detection and Handling

Text encoding represents one of the most complex challenges in file processing, as files rarely declare their encoding explicitly. The component must make intelligent decisions about how to interpret byte sequences while providing fallback strategies when detection fails.

> **Decision: UTF-8 First, Latin-1 Fallback Strategy**
>
> - **Context**: Files can use dozens of different encodings, but detection libraries are complex dependencies that may not be available in all target languages. Many diff tools fail entirely on encoding mismatches.
> - **Options Considered**:
>   1. Require explicit encoding specification via command-line flags
>   2. Use comprehensive encoding detection library (chardet/uchardet)
>   3. Try UTF-8 first, fallback to Latin-1, fail on other encodings
> - **Decision**: Implement UTF-8-first with Latin-1 fallback strategy
> - **Rationale**: UTF-8 handles 90%+ of modern text files and is backward-compatible with ASCII. Latin-1 can represent any byte sequence without errors, providing a universal fallback. This covers the vast majority of real-world scenarios while keeping dependencies minimal.
> - **Consequences**: Enables robust handling of common encodings while maintaining simplicity. May misinterpret files using other encodings (like Shift-JIS or UTF-16), but these represent edge cases for a learning project.

The encoding detection process follows a systematic approach designed to maximize success while minimizing complexity:

1. **UTF-8 Validation Attempt**: Read the entire file as bytes and attempt UTF-8 decoding. UTF-8 has strict byte sequence rules, so invalid sequences will trigger decode errors immediately.
2. **Latin-1 Fallback**: If UTF-8 decoding fails, retry with Latin-1 encoding. Latin-1 assigns meaning to every possible byte value (0-255), so this step cannot fail.
3. **Binary File Detection**: During encoding attempts, detect likely binary files by scanning for null bytes or high percentages of non-printable characters. Binary files should be rejected early with clear error messages.
4. **Encoding Metadata Preservation**: Store the detected encoding in the `FileContent` structure for debugging and potential future use in output formatting.

Detection Step	Trigger Condition	Action Taken	Error Handling
UTF-8 Attempt	Initial file read	Decode entire file as UTF-8	UnicodeDecodeError → proceed to fallback
Binary Detection	During UTF-8 attempt	Scan for null bytes, control chars	Reject with "binary file" error
Latin-1 Fallback	UTF-8 decode failure	Re-read file with Latin-1	Cannot fail (covers all byte values)
Success Recording	Successful decode	Store encoding in FileContent	None

> **Critical Insight**: The encoding detection strategy prioritizes predictable behavior over perfect accuracy. It's better to consistently handle 95% of files correctly than to sometimes handle 100% of files but fail unpredictably on edge cases.

Line Normalization

Once file content is successfully decoded to Unicode strings, the component must address the complexities of line ending conventions and whitespace handling. Different operating systems, text editors, and historical contexts have produced a variety of line ending formats that must be reconciled for accurate comparison.

The normalization process preserves the logical structure of the file while standardizing physical representation details that should not affect diff computation. This requires careful balance between normalization and preservation — we want to eliminate spurious differences while maintaining meaningful distinctions.

Line Ending Standardization

Line endings present a classic compatibility challenge in text processing. The component must detect and normalize different line ending conventions while preserving the original format information for potential restoration.

Line Ending Type	Byte Sequence	Origin	Detection Strategy
Unix LF	`\n` (0x0A)	Unix, Linux, macOS	Single byte 0x0A
Windows CRLF	`\r\n` (0x0D 0x0A)	Windows, DOS	Two-byte sequence, must check \r\n together
Classic Mac CR	`\r` (0x0D)	Pre-OSX Mac	Single byte 0x0D not followed by 0x0A
Mixed Endings	Various combinations	Corrupted or manually edited files	Count each type, report inconsistency

The normalization algorithm processes line endings in a specific order to avoid misdetection:

1. **CRLF Detection First**: Scan for `\r\n` sequences and replace with internal line separator markers. This must happen before individual `\r` or `\n` processing to avoid splitting CRLF pairs.
2. **Individual CR/LF Processing**: After CRLF removal, remaining `\r` or `\n` characters represent Mac or Unix line endings respectively.
3. **Consistency Validation**: Track the types of line endings found and warn if multiple types appear in the same file, as this often indicates file corruption or manual editing errors.
4. **Original Format Preservation**: Store the detected line ending type in `FileContent.original_endings` for potential use in output formatting or debugging.

Whitespace and Empty Line Handling

Whitespace handling requires nuanced decisions about what constitutes meaningful content versus formatting artifacts. The component must preserve structural elements like empty lines while potentially normalizing trailing whitespace that rarely carries semantic meaning.

> **Decision: Preserve Empty Lines, Optional Trailing Whitespace Normalization**
>
> - **Context**: Empty lines often carry semantic meaning (paragraph breaks, logical sections), but trailing whitespace is usually incidental formatting that creates spurious diff noise.
> - **Options Considered**:
>   1. Preserve all whitespace exactly as found in files
>   2. Strip all trailing whitespace from every line
>   3. Configurable whitespace normalization with sensible defaults
> - **Decision**: Always preserve empty lines, provide optional trailing whitespace normalization (default: enabled)
> - **Rationale**: Empty lines structure documents and should be preserved for meaningful diffs. Trailing whitespace rarely carries meaning and often creates noise when files are edited with different editors. Making it configurable allows power users to override when needed.
> - **Consequences**: Reduces spurious whitespace diffs while maintaining document structure. May occasionally hide meaningful trailing whitespace, but this represents an edge case.

The whitespace normalization process operates on each line individually after line ending processing:

1. **Empty Line Preservation**: Lines containing only whitespace are preserved as truly empty lines (`""`) to maintain document structure.
2. **Trailing Whitespace Handling**: Based on configuration, either preserve or strip whitespace from the end of non-empty lines.
3. **Leading Whitespace Preservation**: Always preserve leading whitespace as it commonly carries meaning (indentation, alignment).
4. **Tab/Space Normalization**: Preserve tabs and spaces exactly as found, as the choice between them often carries semantic meaning or project-specific conventions.

Whitespace Type	Handling Strategy	Rationale
Leading spaces/tabs	Always preserve	Carries semantic meaning (indentation)
Trailing spaces/tabs	Configurable strip (default: remove)	Usually formatting artifacts
Empty lines (whitespace-only)	Convert to truly empty	Preserves structure, eliminates noise
Internal spaces/tabs	Always preserve	Part of content, never normalize

Common Pitfalls

The File Reader Component encounters several categories of errors that can cascade into confusing failures in downstream components. Understanding these pitfalls helps implementers build robust error handling and provide clear diagnostic messages.

⚠️ Pitfall: Binary File Encoding Explosion

Problem: Attempting to decode binary files (executables, images, compressed archives) as text causes encoding errors that propagate through the entire diff pipeline, often manifesting as cryptic Unicode exceptions far from the actual problem.

Why It Happens: Binary files contain byte sequences that violate UTF-8 encoding rules or produce invalid Unicode code points. When these sequences reach the LCS algorithm, they can cause comparison failures or generate nonsensical diff output.

Detection Strategy: Scan file content during encoding detection for binary indicators: null bytes (`\0`), high percentages of non-printable characters, or known binary file signatures (magic numbers).

Prevention: Implement binary detection early in the encoding process and fail fast with clear error messages like "Cannot diff binary file: detected null bytes at position X".

⚠️ Pitfall: Memory Exhaustion on Large Files

Problem: Loading entire large files into memory for line processing can exhaust available RAM, causing the process to crash or swap heavily, especially when processing log files or data dumps.

Why It Happens: The naive approach reads the complete file content into a string, then splits into lines, temporarily holding both the original content and the line array in memory simultaneously.

Detection Strategy: Monitor file sizes during initial file stat operations and warn when files exceed reasonable thresholds (e.g., 100MB).

Mitigation: Implement streaming line reading that processes files incrementally, or provide configuration options for memory limits with clear error messages when exceeded.

⚠️ Pitfall: Trailing Newline Inconsistency

Problem: Different text editors handle final newlines inconsistently — some always add a trailing newline, others preserve files exactly as edited. This creates spurious diffs where files differ only in their final newline presence.

Why It Happens: POSIX defines text files as sequences of lines where each line ends with a newline, but many editors don't enforce this. When one file ends with a newline and another doesn't, the line count differs, causing alignment issues in the LCS algorithm.

Detection Strategy: Check whether file content ends with a line ending sequence and track this information separately from line content.

Handling Options: Either normalize by always ensuring a final newline, or preserve the distinction but mark it clearly in diff output with messages like "No newline at end of file".

⚠️ Pitfall: Line Ending Mixed Mode Corruption

Problem: Files containing mixed line endings (some lines end with LF, others with CRLF) can indicate file corruption or editing across different systems. Naive splitting creates inconsistent line parsing where some lines retain carriage returns.

Why It Happens: Files transferred between systems without proper conversion, or edited with tools that don't normalize line endings consistently.

Detection Strategy: Count occurrences of each line ending type during normalization and flag files with multiple types.

Resolution: Provide clear warnings about mixed line endings and document the normalization strategy used, allowing users to fix their files if needed.

Error Category	Early Detection Signal	Recommended Response
Binary files	Null bytes, high non-printable ratio	Fail fast with clear error message
Large files	File size > memory threshold	Warn user, implement streaming if possible
Encoding failures	UnicodeDecodeError in both UTF-8 and Latin-1	Report unsupported encoding, suggest alternatives
Mixed line endings	Multiple line ending types detected	Warn user, document normalization applied
Trailing newline inconsistency	One file ends with newline, other doesn't	Mark in diff output, consider normalization option

Implementation Guidance

The File Reader Component bridges the gap between raw file system operations and the structured data required for diff computation. The implementation must balance robust error handling with clean abstractions that hide complexity from downstream components.

Technology Recommendations

Component	Simple Option	Advanced Option
File I/O	Built-in file operations (`open`, `read`)	Memory-mapped files for large file optimization
Encoding Detection	Try UTF-8, fallback to Latin-1	`chardet` / `uchardet` library for comprehensive detection
Line Splitting	String `split` with line ending normalization	Streaming line reader with configurable buffers
Error Handling	Exception propagation with context	Structured error types with recovery suggestions

Recommended File Structure

```
diff-tool/
  src/
    file_reader/
      __init__.py             ← Component interface
      reader.py               ← Core FileReader class
      encoding.py             ← Encoding detection utilities
      normalization.py        ← Line ending and whitespace handling
      errors.py               ← File reading error types
    tests/
      test_file_reader.py     ← Comprehensive test suite
      fixtures/               ← Test files with various encodings
        utf8_unix.txt
        latin1_windows.txt
        binary_file.exe
        mixed_endings.txt
```

Core Data Structures

The File Reader Component works primarily with the `FileContent` structure that encapsulates all information needed for downstream processing:

PYTHON

```python
from dataclasses import dataclass
from typing import List, Optional

@dataclass
class FileContent:
    """Complete representation of a file prepared for diff processing."""
    filepath: str                    # Original file path for error reporting
    lines: List[str]                 # Normalized line content (no line endings)
    line_count: int                  # Total number of lines for validation
    encoding: str                    # Detected encoding (utf-8 or latin-1)
    original_endings: str            # Original line ending style (LF/CRLF/CR/mixed)

def create_file_content(filepath: str, raw_content: bytes,
                        encoding: str, line_ending: str) -> FileContent:
    """Factory function for validated FileContent creation.

    TODO 1: Normalize line endings according to detected type
    TODO 2: Split content into lines while preserving empty lines
    TODO 3: Apply whitespace normalization based on configuration
    TODO 4: Validate line count matches actual content
    TODO 5: Return populated FileContent with all metadata
    """
    pass
```

Encoding Detection Infrastructure

```python
import os

from typing import Tuple


def detect_file_encoding(filepath: str) -> str:

    """Determine file encoding trying UTF-8 then Latin-1.



    Returns the encoding name that successfully decoded the file.

    Raises BinaryFileError if file appears to be binary.

    """

    # TODO 1: Read file as bytes for encoding detection

    # TODO 2: Check for binary file indicators (null bytes, high non-printable ratio)

    # TODO 3: Attempt UTF-8 decoding on entire content

    # TODO 4: On UTF-8 failure, attempt Latin-1 decoding

    # TODO 5: Return successful encoding name

    pass


def _is_binary_content(content: bytes, sample_size: int = 8192) -> bool:

    """Detect if content appears to be binary rather than text.



    TODO 1: Check for null bytes in content sample

    TODO 2: Count printable vs non-printable characters

    TODO 3: Return True if binary indicators exceed threshold

    """

    pass


def read_file_lines(filepath: str) -> List[str]:

    """Read file with encoding detection and line normalization.
```

```
    TODO 1: Validate file exists and is readable

    TODO 2: Detect encoding using detect_file_encoding()

    TODO 3: Read content with detected encoding

    TODO 4: Normalize line endings and split into lines

    TODO 5: Apply whitespace normalization

    TODO 6: Return clean list of line strings

    """

    pass
```

Line Normalization Utilities

```python
from typing import Tuple, List

import re


def normalize_line_endings(content: str) -> Tuple[str, str]:

    """Normalize line endings and detect original format.



    Returns normalized content and detected line ending type.

    """

    # TODO 1: Count occurrences of CRLF, LF, and CR

    # TODO 2: Determine predominant line ending style

    # TODO 3: Warn if mixed line endings detected

    # TODO 4: Convert all line endings to LF for internal processing

    # TODO 5: Return normalized content and original ending type

    pass


def split_preserving_empty_lines(content: str) -> List[str]:

    """Split content into lines while preserving empty lines.



    TODO 1: Split on normalized line endings (LF)

    TODO 2: Handle final newline correctly (don't create extra empty line)

    TODO 3: Preserve truly empty lines in sequence

    TODO 4: Return list of line content without line ending characters

    """

    pass


def normalize_whitespace(lines: List[str], strip_trailing: bool = True) -> List[str]:

    """Apply whitespace normalization to line list.
```

```
    TODO 1: Process each line individually

    TODO 2: Preserve leading whitespace (semantic meaning)

    TODO 3: Optionally strip trailing whitespace based on configuration

    TODO 4: Convert whitespace-only lines to empty strings

    TODO 5: Return normalized line list

    """

    pass
```

Error Handling and Recovery

PYTHON

```python
class FileReaderError(Exception):

    """Base exception for file reading operations."""

    pass


class BinaryFileError(FileReaderError):

    """Raised when attempting to process binary files as text."""

    def __init__(self, filepath: str, indicator: str):

        super().__init__(f"Cannot diff binary file {filepath}: {indicator}")


class EncodingDetectionError(FileReaderError):

    """Raised when file encoding cannot be determined."""

    def __init__(self, filepath: str, attempted_encodings: List[str]):

        encodings = ", ".join(attempted_encodings)

        super().__init__(f"Cannot detect encoding for {filepath}, tried: {encodings}")


class LargeFileError(FileReaderError):

    """Raised when file exceeds memory or processing limits."""

    def __init__(self, filepath: str, size: int, limit: int):

        super().__init__(f"File {filepath} ({size} bytes) exceeds limit ({limit} bytes)")
```

Milestone Checkpoint

After implementing the File Reader Component, verify correct behavior with these test scenarios:

Test Command:

```bash
python -m pytest tests/test_file_reader.py -v
```

Expected Behavior:

- UTF-8 files read correctly with proper line splitting
- Latin-1 files fall back gracefully when UTF-8 fails
- Binary files rejected with clear error messages
- Mixed line endings detected and normalized consistently
- Empty lines preserved in output
- Trailing whitespace handled according to configuration

Manual Verification:

```python
from file_reader import read_file_lines

# Test various file types
utf8_lines = read_file_lines('test_files/utf8_sample.txt')
latin1_lines = read_file_lines('test_files/latin1_sample.txt')
print(f"UTF-8 file: {len(utf8_lines)} lines")
print(f"Latin-1 file: {len(latin1_lines)} lines")

# Verify empty line preservation
if "" in utf8_lines:
    print("✓ Empty lines preserved")
```

Debugging Signs:

- **Files appear to have wrong line counts**: Check line ending normalization — mixed CRLF/LF can cause double-counting
- **Unicode decode errors**: Verify binary file detection is working and catching non-text files early
- **Memory usage spikes**: Implement file size checking and streaming for large files
- **Spurious whitespace diffs**: Confirm trailing whitespace normalization is applied consistently

LCS Engine Component

> **Milestone(s):** Milestone 2: LCS Algorithm — implements the core dynamic programming algorithm for finding longest common subsequences, which forms the foundation for all diff generation in subsequent milestones

The LCS Engine represents the mathematical heart of our diff tool, where we transform the abstract problem of "what changed between two files" into a concrete algorithmic solution. This component encapsulates the dynamic programming approach to finding the Longest Common Subsequence, serving as the bridge between raw line data from the File Reader and meaningful diff operations for the Diff Generator.

Mental Model: Finding Common Ground

Think of the LCS algorithm like finding the longest sequence of topics two people agree on during a complex conversation. Imagine Alice and Bob are discussing a project, and you're taking notes on their areas of agreement. Alice says: "We need planning, design, implementation, testing, deployment." Bob says: "We should do planning, prototyping, implementation, review, testing, launch."

Your job is to find the longest sequence of topics they both mentioned in the same order. You can't rearrange their words, but you can skip disagreements to find the common thread. The result might be: "planning, implementation, testing" — a subsequence that appears in both conversations while preserving the original order.

The LCS algorithm works similarly with text lines. Given two files, it finds the longest sequence of lines that appear in both files in the same relative order. These common lines represent the unchanged "skeleton" of the document, while everything else represents insertions or deletions. This common skeleton becomes the foundation for generating meaningful diff output that humans can understand.

The power of this approach lies in its optimality guarantee. Among all possible ways to align two sequences, the LCS algorithm finds the alignment that maximizes preserved content and minimizes the number of changes needed. This mathematical property ensures our diff output shows the most intuitive representation of what actually changed between files.

Dynamic Programming Algorithm

The LCS algorithm employs dynamic programming to solve what would otherwise be an exponentially complex problem. The core insight is that the LCS of two sequences can be computed by solving smaller subproblems and combining their results systematically.

> **Decision: Dynamic Programming Matrix Approach**
>
> - **Context**: Multiple algorithms exist for computing LCS, including recursive, memoized recursive, and dynamic programming approaches
> - **Options Considered**:
>   - Naive recursive: Simple but exponential time complexity O(2^n)
>   - Memoized recursive: Better performance but complex stack management
>   - Dynamic programming matrix: O(mn) time and space with clear implementation
> - **Decision**: Use bottom-up dynamic programming with explicit matrix construction
> - **Rationale**: Provides predictable O(mn) performance, easier debugging through matrix inspection, and straightforward backtracking for sequence recovery
> - **Consequences**: Higher memory usage for large files but excellent performance characteristics and implementation clarity for learning purposes

Algorithm Approach	Time Complexity	Space Complexity	Debuggability	Implementation Complexity
Naive Recursive	O(2^n)	O(n) stack	Poor	Simple
Memoized Recursive	O(mn)	O(mn) + stack	Medium	Medium
DP Matrix (Chosen)	**O(mn)**	**O(mn)**	**Excellent**	**Medium**

The dynamic programming solution builds a two-dimensional matrix where each cell `[i][j]` represents the length of the LCS for the first `i` elements of sequence 1 and the first `j` elements of sequence 2. This bottom-up approach systematically solves smaller subproblems to construct the solution for the full problem.

Matrix Construction Algorithm

The matrix construction follows a systematic pattern that builds the LCS length table:

1. **Initialize the matrix** with dimensions `(len(sequence1) + 1) × (len(sequence2) + 1)` to accommodate empty sequence cases. The extra row and column represent empty prefixes with LCS length zero.

2. **Set base cases** by filling the first row and first column with zeros, representing that the LCS of any sequence with an empty sequence has length zero.

3. **Fill the matrix iteratively** using the recurrence relation. For each cell `[i][j]`, compare `sequence1[i-1]` with `sequence2[j-1]` (subtracting 1 because matrix indices are offset by 1).

4. **Apply the recurrence relation**: If the elements match, set `matrix[i][j] = matrix[i-1][j-1] + 1` (extending the LCS of the prefixes). If they don't match, set `matrix[i][j] = max(matrix[i-1][j], matrix[i][j-1])` (taking the better of excluding one element or the other).

5. **Complete the matrix** by processing all cells in row-major order, ensuring each cell depends only on previously computed values.

6. **Extract the LCS length** from the bottom-right cell `matrix[len(sequence1)][len(sequence2)]`, which represents the LCS length for the complete sequences.

The recurrence relation captures the essential logic of LCS computation. When elements match, we can extend the best solution for the shorter prefixes. When they don't match, we take the better of two options: either exclude the current element from sequence 1 or exclude it from sequence 2. This greedy choice at each step leads to the globally optimal solution due to the optimal substructure property.

Backtracking Algorithm for Sequence Recovery

Building the matrix gives us the LCS length, but we need the actual sequence for diff generation. The backtracking algorithm reconstructs the LCS by tracing backwards through the matrix:

1. **Start at the bottom-right corner** `[len(sequence1)][len(sequence2)]` where the complete LCS length is stored.
2. **Initialize tracking variables** for the current matrix position `(i, j)` and an empty list to collect LCS elements in reverse order.
3. **Trace backwards through the matrix** by examining how each cell was computed. If `sequence1[i-1] == sequence2[j-1]`, this element is part of the LCS, so add it to the result and move diagonally to `[i-1][j-1]`.
4. **Handle non-matching elements** by moving toward the cell that contributed the maximum value. If `matrix[i-1][j] > matrix[i][j-1]`, move up to `[i-1][j]`. Otherwise, move left to `[i][j-1]`.
5. **Continue until reaching the top or left edge** where the matrix values are zero, indicating we've traced back to empty sequence prefixes.
6. **Reverse the collected elements** since backtracking produces them in reverse order, yielding the actual LCS.

The backtracking process also captures the positions where LCS elements occur in both original sequences, which becomes crucial for diff generation. By tracking the `(i-1, j-1)` coordinates when we find matching elements, we build a mapping showing where each common line appears in both files.

LCS Engine Data Structures

The `LCSEngine` component manages the dynamic programming computation and provides a clean interface for the rest of the system:

Field	Type	Description
`matrix`	`list[list[int]]`	The dynamic programming table storing LCS lengths for all subproblems
`sequence1`	`Sequence`	The first input sequence (typically lines from file 1)
`sequence2`	`Sequence`	The second input sequence (typically lines from file 2)
`lcs_length`	`int`	The length of the computed LCS, cached after matrix construction
`computation_stats`	`dict`	Performance metrics including matrix size and computation time

The `CommonSubsequence` structure encapsulates the complete LCS result:

Field	Type	Description
`elements`	`list[str]`	The actual LCS elements in sequence order
`length`	`int`	The number of elements in the LCS
`positions1`	`list[int]`	Zero-based indices where LCS elements appear in sequence1
`positions2`	`list[int]`	Zero-based indices where LCS elements appear in sequence2

The position lists enable the Diff Generator to identify exactly which lines are unchanged and where they occur in both files, forming the anchor points around which insertions and deletions are organized.

LCS Engine Interface

The `LCSEngine` exposes a focused interface that encapsulates the dynamic programming complexity:

Method	Parameters	Returns	Description
`compute_lcs`	`sequence1: Sequence, sequence2: Sequence`	`CommonSubsequence`	Main entry point that builds matrix and extracts LCS
`build_lcs_matrix`	`seq1: Sequence, seq2: Sequence`	`list[list[int]]`	Constructs the dynamic programming matrix
`backtrack`	`matrix: list[list[int]], seq1: Sequence, seq2: Sequence`	`CommonSubsequence`	Recovers the actual LCS from the completed matrix
`get_matrix_cell`	`i: int, j: int`	`int`	Safe matrix access with bounds checking
`clear_matrix`	`None`	`None`	Releases matrix memory after computation

Memory Optimization Strategies

The standard LCS dynamic programming approach requires O(mn) space for the matrix, which becomes problematic for large files. A 10,000-line file compared against another 10,000-line file requires a 100-million-cell matrix, consuming significant memory.

> **Decision: Adaptive Memory Strategy**
>
> - **Context**: Large files can exhaust available memory with O(mn) matrix storage, but different use cases have different memory constraints
> - **Options Considered**:
>   - Always use O(mn) matrix: Simple but fails on large files
>   - Always use Hirschberg's algorithm: O(n) space but complex implementation
>   - Adaptive approach: Choose strategy based on input size
> - **Decision**: Implement adaptive memory management with configurable thresholds
> - **Rationale**: Provides optimal performance for small-medium files while gracefully handling large files through space-efficient algorithms
> - **Consequences**: More complex implementation but handles full range of input sizes from small config files to large source files

Strategy	Space Complexity	Implementation Complexity	Performance	File Size Limit
Standard Matrix (Chosen for <1M cells)	**O(mn)**	**Low**	**Best**	**~1000×1000 lines**
Two-Row Optimization	**O(min(m,n))**	**Medium**	**Good**	**~10K×10K lines**
Hirschberg's Algorithm	**O(m+n)**	**High**	**Slower**	**Unlimited**

Two-Row Optimization

For computing just the LCS length without sequence recovery, we can reduce space complexity to O(min(m,n)) by observing that each matrix row depends only on the previous row. This optimization maintains two arrays instead of the full matrix:

The algorithm maintains `current_row` and `previous_row` arrays, updating them as it processes each row of the conceptual matrix. After processing each row, the roles swap: `current_row` becomes `previous_row` for the next iteration. This approach reduces memory usage by a factor of max(m,n) while maintaining the same time complexity.

However, this optimization complicates backtracking since we no longer have the full matrix. For diff tools, we need the actual LCS sequence, not just its length, so this optimization applies primarily to preliminary size checking or when combined with more sophisticated space-efficient algorithms.

Hirschberg's Algorithm Consideration

Hirschberg's algorithm achieves O(m+n) space complexity while still recovering the actual LCS through a divide-and-conquer approach. The algorithm recursively splits the problem in half, using the two-row optimization to find the optimal splitting point, then recursively processes each half.

For our diff tool, Hirschberg's algorithm becomes relevant when comparing very large files that exceed available memory with the standard matrix approach. The implementation complexity significantly increases, involving recursive partitioning and careful coordinate tracking across subproblems.

> The key insight is that Hirschberg's algorithm trades time for space — it performs multiple passes over the data to avoid storing the complete matrix, resulting in roughly double the computation time but dramatically reduced memory usage.

Adaptive Memory Management Implementation

Our LCS engine implements adaptive memory management by estimating memory requirements before choosing the algorithm:

1. **Calculate matrix size requirements** by multiplying sequence lengths and estimating memory per cell (typically 4-8 bytes for integers).

2. **Compare against available memory** using system memory detection and configured memory limits for the diff process.
3. **Select algorithm based on thresholds**: Use standard matrix for small files, two-row optimization for medium files, and Hirschberg's algorithm for large files that would exceed memory limits.
4. **Provide progress feedback** for large file processing, since space-efficient algorithms take longer and users need feedback on processing status.
5. **Implement graceful degradation** by falling back to simpler diff approaches (like line-by-line comparison without LCS optimization) if even the space-efficient algorithms encounter memory pressure.

Common Pitfalls

Understanding the typical mistakes in LCS implementation helps avoid debugging sessions and ensures robust diff functionality.

⚠️ Pitfall: Off-by-One Matrix Indexing

The most frequent LCS implementation error involves matrix indexing confusion between sequence positions and matrix positions. The matrix has dimensions `(m+1) × (n+1)` to include empty sequence cases, but sequence elements are accessed using 0-based indexing.

When filling matrix cell `[i][j]`, the corresponding sequence elements are `sequence1[i-1]` and `sequence2[j-1]`. Forgetting this offset leads to index out-of-bounds errors or comparing wrong elements. The bug manifests as either runtime crashes on boundary cases or subtly incorrect LCS results where elements appear to match when they shouldn't.

Fix: Always use `sequence1[i-1]` and `sequence2[j-1]` when filling matrix cell `[i][j]`, and carefully trace through boundary cases with empty sequences to verify indexing logic.

⚠️ Pitfall: Empty Sequence Handling

Empty files or sequences with no matching elements create edge cases that break poorly implemented LCS algorithms. Common issues include division by zero when calculating progress percentages, null pointer exceptions when accessing empty sequence elements, or infinite loops in backtracking when no valid LCS exists.

The algorithm should handle empty inputs gracefully: empty sequences have LCS length zero, and backtracking should immediately return an empty result. Additionally, sequences with no common elements should produce an empty LCS, not crash the algorithm.

Fix: Explicitly test with empty sequences and implement early returns for trivial cases. Add null checks and boundary validation before accessing sequence elements or matrix cells.

⚠️ Pitfall: Memory Explosion on Large Files

Large files can quickly exhaust available memory without appropriate safeguards. A 50,000-line file compared against another similar file requires a 2.5-billion-cell matrix, consuming 10+ GB of memory with standard integer storage.

Without memory limits or adaptive algorithms, the diff tool becomes unusable on realistic large files like database dumps, log files, or generated code. The system may freeze, swap thrash, or crash with out-of-memory errors, providing no useful feedback to the user.

Fix: Implement memory estimation before matrix allocation, provide user feedback about memory requirements, and implement fallback algorithms for large files. Consider streaming approaches or approximate diff algorithms when exact LCS computation exceeds practical memory limits.

⚠️ Pitfall: Incorrect Backtracking Logic

Backtracking errors produce incorrect LCS sequences even when the matrix construction is correct. Common mistakes include wrong direction choices when matrix values are equal, failing to handle diagonal moves correctly, or reconstructing the sequence in the wrong order.

These bugs are particularly insidious because the LCS length might be correct while the actual sequence is wrong, leading to malformed diffs that incorrectly identify changed lines. The resulting diff output becomes unreliable and may show spurious additions or deletions.

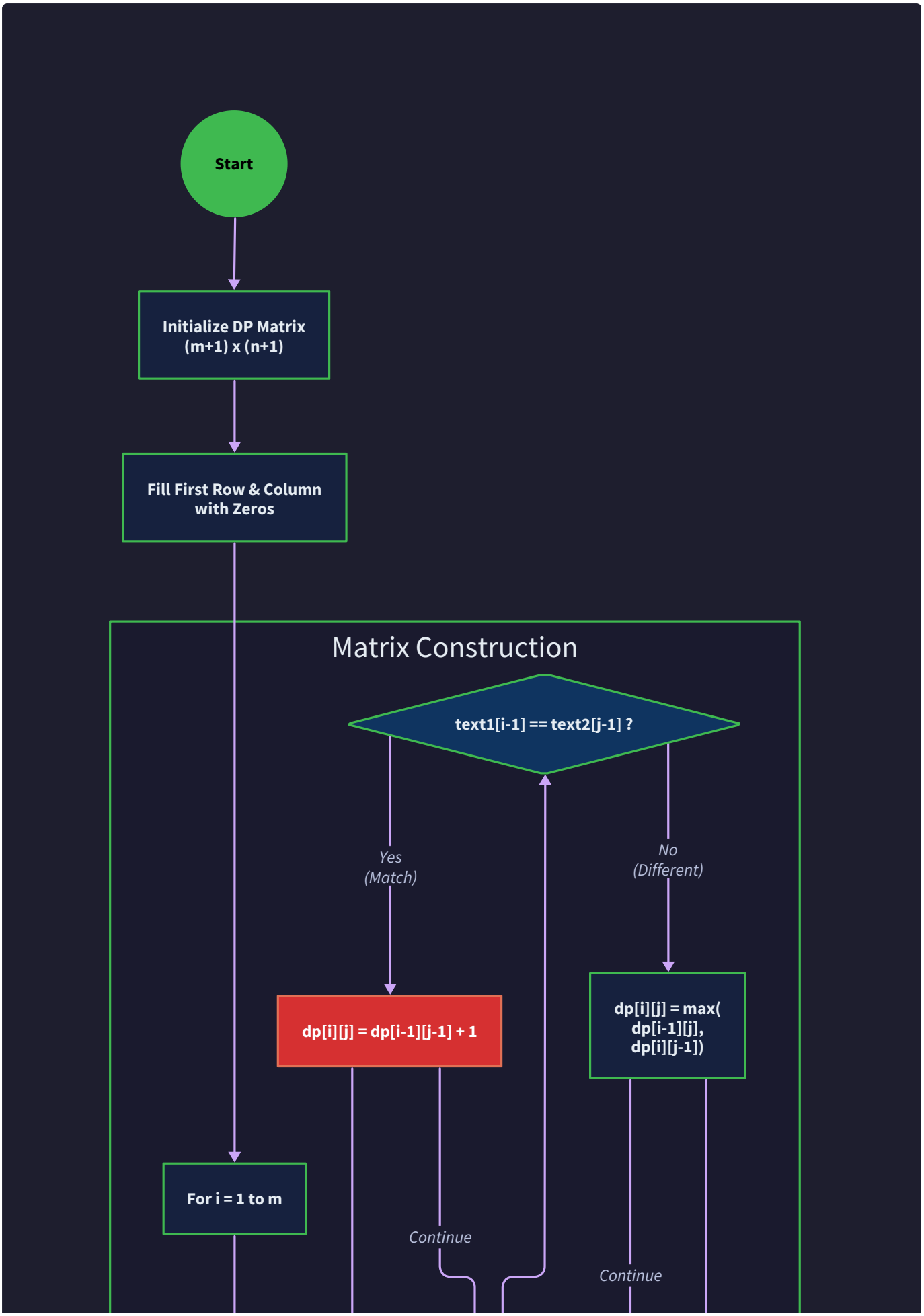
Fix: Implement comprehensive backtracking tests with known input-output pairs, carefully handle tie-breaking when multiple cells have equal values, and verify that backtracking produces sequences that actually appear in both original inputs.

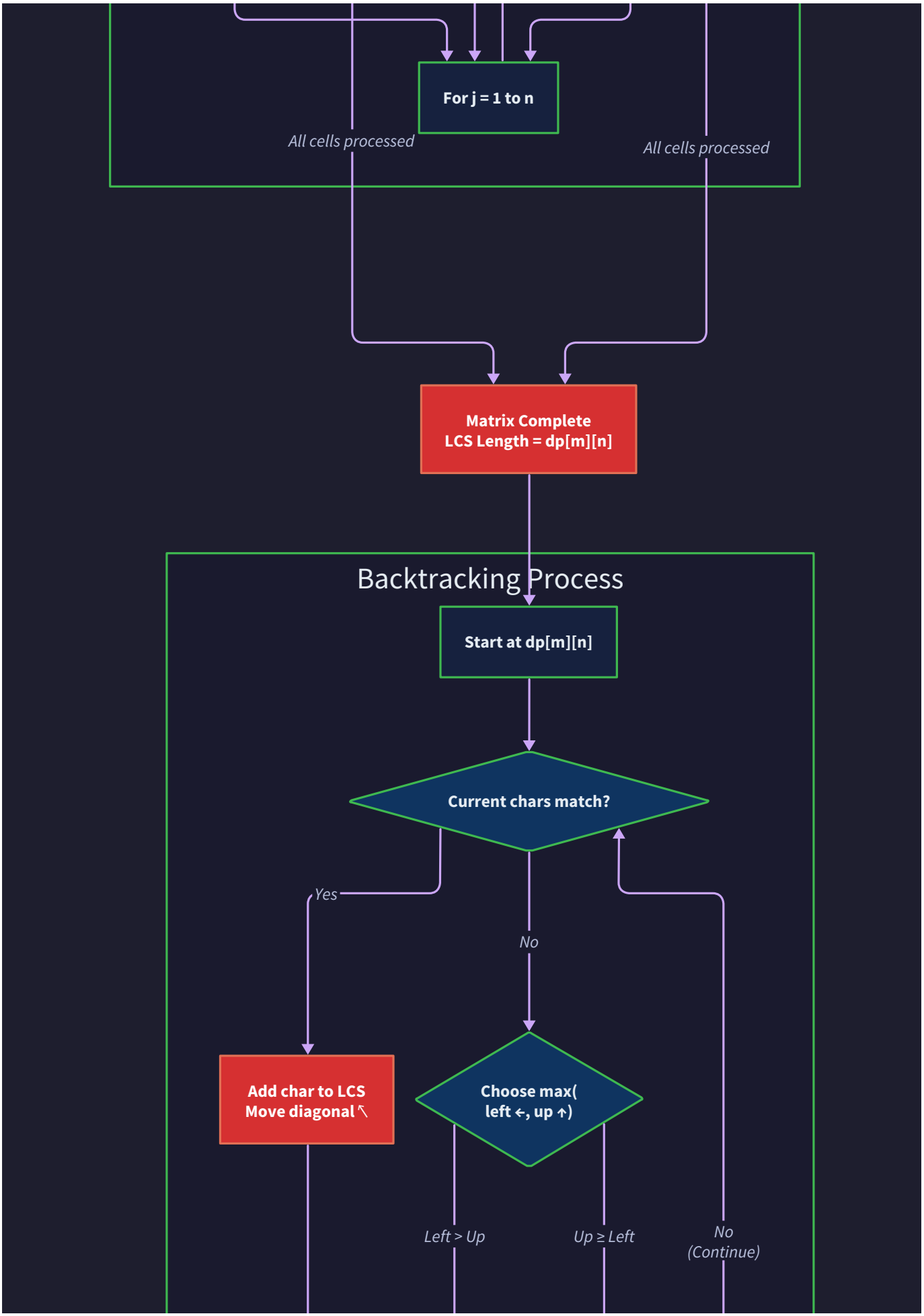
⚠️ Pitfall: Performance Degradation with Repetitive Content

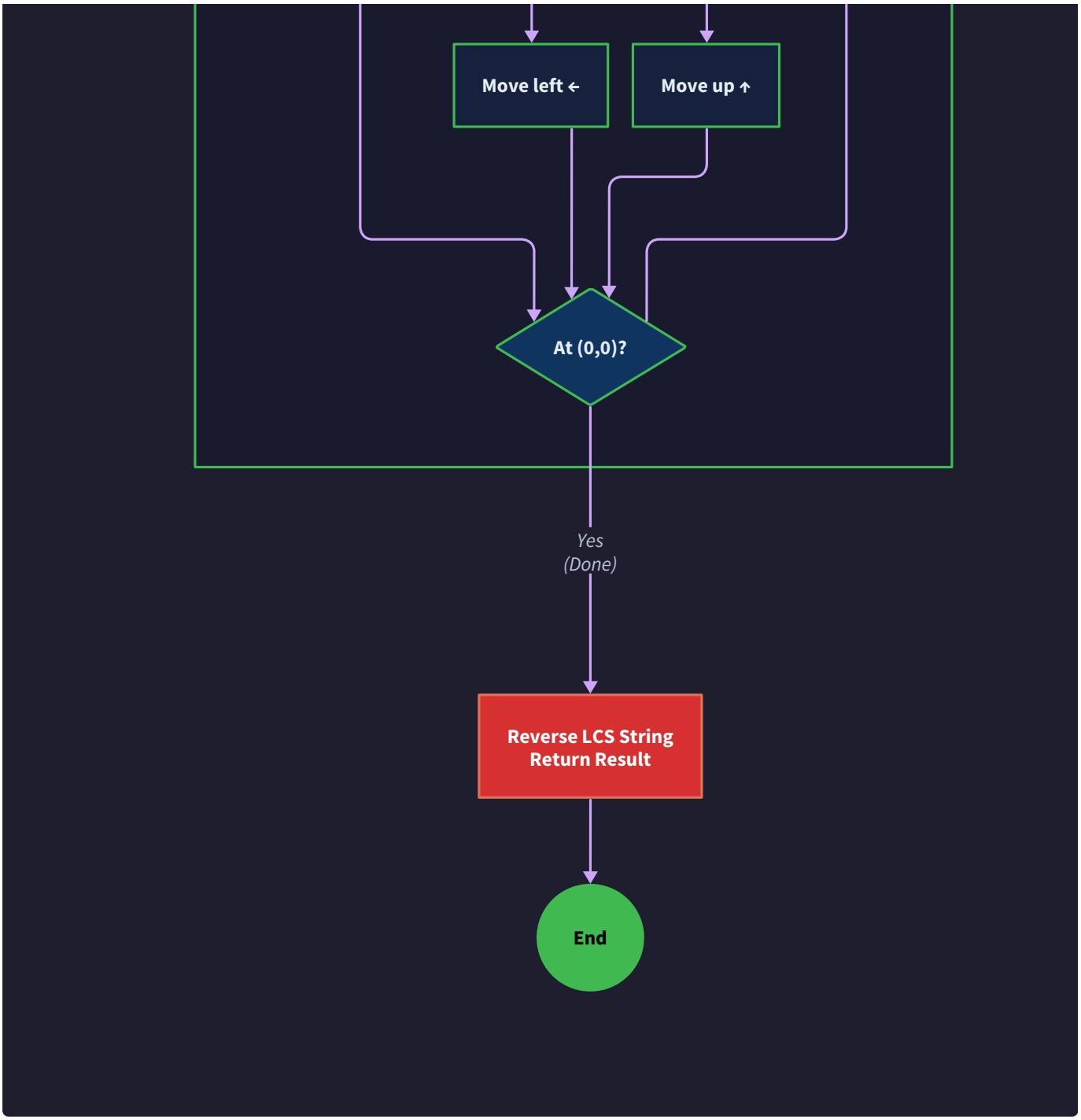
Files with highly repetitive content (like configuration templates, generated code, or data dumps) create performance challenges for LCS algorithms. Many potential matches exist at each step, leading to large matrices with complex backtracking paths.

While the algorithm complexity remains O(mn), the constant factors become significant, and cache performance degrades due to poor locality of access patterns. The user experiences slow performance even on moderately-sized files with repetitive structure.

Fix: Implement preprocessing to identify and handle repetitive patterns efficiently, consider approximate algorithms for files with low diversity, and provide progress feedback for long-running computations on repetitive content.


Start
Initialize DP Matrix (m+1) x (n+1)
Fill First Row & Column with Zeros
Matrix Construction
text1[i-1] == text2[j-1] ?
Yes (Match)
No (Different)
dp[i][j] = dp[i-1][j-1] + 1
dp[i][j] = max(dp[i-1][j], dp[i][j-1])
For i = 1 to m
Continue
Continue


For j = 1 to n
All cells processed
All cells processed
Matrix Complete
LCS Length = dp[m][n]
Backtracking Process
Start at dp[m][n]
Current chars match?
Yes
No
Add char to LCS
Move diagonal ↖
Choose max(
left ←, up ↑)
Left > Up
Up ≥ Left
No
(Continue)



Implementation Guidance

Technology Recommendations

Component	Simple Option	Advanced Option
Matrix Storage	`list[list[int]]` (Python native lists)	`numpy.ndarray` for large matrices with better memory layout
Sequence Representation	`list[str]` (Python native strings)	Custom `Sequence` class with lazy loading for large files

Component	Simple Option	Advanced Option
Memory Management	Manual size checking with `sys.getsizeof()`	Memory-mapped files with `mmap` for huge sequences
Performance Monitoring	Simple time tracking with `time.perf_counter()`	`cProfile` integration for detailed algorithm profiling

Recommended File Structure

The LCS Engine fits into the project structure as a focused algorithmic component:

```
diff_tool/
├── core/
│   ├── __init__.py
│   ├── lcs_engine.py          ← main LCS implementation
│   ├── lcs_types.py           ← CommonSubsequence and related types
│   └── memory_optimizer.py    ← adaptive memory management
├── utils/
│   ├── sequence_utils.py      ← sequence preprocessing helpers
│   └── performance_monitor.py ← computation timing and stats
└── tests/
    ├── test_lcs_engine.py     ← comprehensive LCS algorithm tests
    ├── test_lcs_edge_cases.py ← empty sequences, large files, etc.
    └── fixtures/              ← test data files with known LCS results
```

Infrastructure Starter Code

Complete Memory Management Helper (`core/memory_optimizer.py`):

PYTHON

```python
import sys

import psutil

from typing import Tuple, Literal

from dataclasses import dataclass


@dataclass

class MemoryStrategy:

    algorithm: Literal["matrix", "two_row", "hirschberg"]

    max_memory_mb: float

    estimated_time_factor: float


class MemoryOptimizer:

    """Determines optimal LCS algorithm based on available memory and input size."""



    def __init__(self, max_memory_percent: float = 0.5):

        self.max_memory_percent = max_memory_percent

        available_memory = psutil.virtual_memory().available

        self.max_memory_bytes = available_memory * max_memory_percent



    def choose_strategy(self, seq1_len: int, seq2_len: int) -> MemoryStrategy:

        """Choose LCS algorithm based on memory requirements and available resources."""

        matrix_cells = (seq1_len + 1) * (seq2_len + 1)

        matrix_memory = matrix_cells * 8  # 8 bytes per int64



        if matrix_memory <= self.max_memory_bytes:

            return MemoryStrategy("matrix", matrix_memory / 1024 / 1024, 1.0)

        elif seq1_len * 8 <= self.max_memory_bytes:  # Two row optimization

            return MemoryStrategy("two_row", seq1_len * 16 / 1024 / 1024, 1.2)

        else:
```

```
            return MemoryStrategy("hirschberg", (seq1_len + seq2_len) * 8 / 1024 / 1024, 2.5)


    def estimate_computation_time(self, seq1_len: int, seq2_len: int, strategy:
MemoryStrategy) -> float:
        """Estimate computation time in seconds based on sequence length and strategy."""
        base_operations = seq1_len * seq2_len
        # Rough estimate: 1M operations per second baseline
        base_time = base_operations / 1_000_000
        return base_time * strategy.estimated_time_factor
```

Complete Performance Monitor (`utils/performance_monitor.py`):

PYTHON

```python
import time

from typing import Dict, Any, Optional

from dataclasses import dataclass, field


@dataclass

class ComputationStats:

    """Statistics collected during LCS computation."""

    start_time: float = field(default_factory=time.perf_counter)

    end_time: Optional[float] = None

    matrix_size: Tuple[int, int] = (0, 0)

    memory_used_mb: float = 0.0

    algorithm_used: str = "matrix"

    lcs_length: int = 0



    @property

    def computation_time(self) -> float:

        if self.end_time is None:

            return time.perf_counter() - self.start_time

        return self.end_time - self.start_time



    def finish(self):

        """Mark computation as complete."""

        self.end_time = time.perf_counter()


class PerformanceMonitor:

    """Tracks performance metrics during LCS computation."""



    def __init__(self):

        self.current_stats: Optional[ComputationStats] = None
```

```
    def start_computation(self, seq1_len: int, seq2_len: int, algorithm: str) ->
ComputationStats:
        """Begin tracking a new LCS computation."""
        self.current_stats = ComputationStats(
            matrix_size=(seq1_len + 1, seq2_len + 1),
            algorithm_used=algorithm
        )
        return self.current_stats

    def finish_computation(self, lcs_length: int) -> ComputationStats:
        """Complete the current computation tracking."""
        if self.current_stats is None:
            raise ValueError("No computation in progress")

        self.current_stats.lcs_length = lcs_length
        self.current_stats.finish()
        return self.current_stats

    def report_progress(self, current_row: int, total_rows: int) -> None:
        """Report progress during matrix construction."""
        if self.current_stats and current_row % 1000 == 0:
            progress = (current_row / total_rows) * 100
            elapsed = time.perf_counter() - self.current_stats.start_time
            print(f"LCS Progress: {progress:.1f}% ({current_row}/{total_rows} rows,
{elapsed:.1f}s)")
```

Core Logic Skeleton Code

Main LCS Engine (`core/lcs_engine.py`):

PYTHON

```python
from typing import List, Optional
from dataclasses import dataclass
from .lcs_types import CommonSubsequence, Sequence
from .memory_optimizer import MemoryOptimizer
from ..utils.performance_monitor import PerformanceMonitor

class LCSEngine:
    """Implements dynamic programming LCS algorithm with adaptive memory management."""

    def __init__(self, max_memory_percent: float = 0.5):
        self.memory_optimizer = MemoryOptimizer(max_memory_percent)
        self.performance_monitor = PerformanceMonitor()
        self.matrix: Optional[List[List[int]]] = None

    def compute_lcs(self, sequence1: Sequence, sequence2: Sequence) -> CommonSubsequence:
        """
        Find the longest common subsequence using dynamic programming.

        This is the main entry point that orchestrates matrix construction,
        backtracking, and memory optimization based on input size.
        """
        # TODO 1: Choose memory strategy based on sequence lengths using memory_optimizer
        # TODO 2: Start performance monitoring for the computation
        # TODO 3: Build the LCS matrix using the chosen strategy
        # TODO 4: Extract the LCS through backtracking
        # TODO 5: Clean up matrix memory and finish performance monitoring
        # TODO 6: Return CommonSubsequence with elements and position mappings
        # Hint: Handle empty sequences as a special case before matrix construction
```

```
        pass

    def build_lcs_matrix(self, seq1: Sequence, seq2: Sequence) -> List[List[int]]:
        """
        Construct the dynamic programming matrix for LCS computation.

        Creates an (m+1) x (n+1) matrix where matrix[i][j] represents
        the LCS length for seq1[:i] and seq2[:j].
        """
        # TODO 1: Create matrix with dimensions (len(seq1)+1) x (len(seq2)+1)
        # TODO 2: Initialize first row and column to zero (base cases)
        # TODO 3: Fill matrix using double loop over sequences
        # TODO 4: For each cell [i,j], compare seq1[i-1] with seq2[j-1]
        # TODO 5: If elements match: matrix[i][j] = matrix[i-1][j-1] + 1
        # TODO 6: If different: matrix[i][j] = max(matrix[i-1][j], matrix[i][j-1])
        # TODO 7: Report progress every 1000 rows for large matrices
        # Hint: Remember the +1 offset between matrix indices and sequence indices
        pass

    def backtrack(self, matrix: List[List[int]], seq1: Sequence, seq2: Sequence) ->
CommonSubsequence:
        """
        Recover the actual LCS from the completed matrix.

        Traces backwards from matrix[m][n] to reconstruct the LCS elements
        and their positions in both original sequences.
        """
        # TODO 1: Initialize tracking variables: i=len(seq1), j=len(seq2)
```

```
        # TODO 2: Create empty lists for LCS elements and position tracking
        # TODO 3: While both i>0 and j>0, examine current matrix cell
        # TODO 4: If seq1[i-1] == seq2[j-1], add to LCS and move diagonally
        # TODO 5: Otherwise, move toward cell that contributed the maximum value
        # TODO 6: Handle remaining elements if one sequence is exhausted first
        # TODO 7: Reverse collected elements (backtracking gives reverse order)
        # TODO 8: Return CommonSubsequence with elements and position arrays
        # Hint: Track positions as (i-1, j-1) when adding matching elements
        pass


    def clear_matrix(self) -> None:
        """Release matrix memory after computation."""
        # TODO 1: Set self.matrix to None to allow garbage collection
        # TODO 2: Optionally call gc.collect() for immediate memory release
        pass
```

LCS Data Types (`core/lcs_types.py`):

```python
from dataclasses import dataclass
from typing import List, Optional

# Type alias for sequence of comparable elements
Sequence = List[str]

@dataclass
class CommonSubsequence:
    """Result of LCS computation with elements and position tracking."""
    elements: List[str]
    length: int
    positions1: List[int]  # Indices in first sequence
    positions2: List[int]  # Indices in second sequence

    def __post_init__(self):
        """Validate that all lists have consistent lengths."""
        # TODO 1: Check that len(elements) == length
        # TODO 2: Check that len(positions1) == len(positions2) == length
        # TODO 3: Raise ValueError if any length mismatches found
        pass

    @property
    def is_empty(self) -> bool:
        """Check if the LCS is empty (no common elements)."""
        return self.length == 0

    def get_element_at_position(self, lcs_index: int) -> tuple[str, int, int]:
        """Get LCS element and its positions in both original sequences."""
        # TODO 1: Validate lcs_index is within bounds
```

```
        # TODO 2: Return tuple of (element, pos1, pos2)
        pass
```

Language-Specific Hints

Python-Specific Optimizations:

- Use `list` comprehensions for matrix initialization: `[[0] * cols for _ in range(rows)]` avoids reference sharing bugs
- Consider `array.array('i')` instead of lists for large matrices to reduce memory overhead
- Use `sys.getsizeof()` to measure actual memory usage of data structures
- Profile with `cProfile` to identify bottlenecks in matrix construction vs. backtracking
- For very large sequences, consider `numpy` arrays with `dtype=np.int32` for better cache performance

Memory Management:

- Monitor memory usage with `psutil.Process().memory_info().rss` during computation
- Implement matrix chunking for files larger than available memory
- Use `gc.collect()` after clearing large matrices to ensure immediate memory release
- Consider `mmap` for reading very large files without loading entirely into memory

Milestone Checkpoint

After implementing the LCS Engine component, verify the following behavior:

Unit Test Verification:

```bash
python -m pytest tests/test_lcs_engine.py -v
```

Expected Test Results:

- `test_empty_sequences`: LCS of empty sequences returns empty CommonSubsequence
- `test_identical_sequences`: LCS of identical sequences returns the full sequence
- `test_no_common_elements`: LCS of completely different sequences returns empty result
- `test_simple_lcs`: Known input-output pairs produce expected LCS elements and positions
- `test_matrix_construction`: Matrix dimensions and values match hand-calculated examples
- `test_backtracking_accuracy`: Backtracking produces sequences that actually exist in both inputs

Manual Verification:

Create test files with known LCS results:

```python
# Test with simple sequences

seq1 = ["A", "B", "C", "D", "E"]

seq2 = ["A", "C", "E", "F"]

# Expected LCS: ["A", "C", "E"] at positions ([0,2,4], [0,1,2])


engine = LCSEngine()

result = engine.compute_lcs(seq1, seq2)

print(f"LCS: {result.elements}")  # Should show ["A", "C", "E"]

print(f"Length: {result.length}") # Should show 3

print(f"Positions: {result.positions1}, {result.positions2}") # Should show [0,2,4], [0,1,2]
```

Performance Verification:

Test memory optimization with large sequences:

```python
# Generate large test sequences

seq1 = [f"line_{i}" for i in range(10000)]

seq2 = [f"line_{i}" if i % 2 == 0 else f"different_{i}" for i in range(10000)]


engine = LCSEngine(max_memory_percent=0.1)  # Restrict memory

result = engine.compute_lcs(seq1, seq2)

# Should complete without memory errors and choose appropriate algorithm
```

Signs of Problems:

Symptom	Likely Cause	Fix
IndexError during matrix construction	Off-by-one in sequence access	Use `seq[i-1]` when filling `matrix[i][j]`
Wrong LCS length but no crash	Incorrect recurrence relation	Verify max() logic and diagonal increment
Empty LCS for sequences with obvious matches	Backtracking logic error	Check direction choices and sequence comparison
Memory errors on medium files	No memory optimization	Implement strategy selection based on input size
Slow performance on repetitive content	Poor cache locality	Consider preprocessing or approximate algorithms

Diff Generator Component

> **Milestone(s):** Milestone 3: Diff Generation — converts LCS results into edit operations and groups them into contextual hunks with unified diff format

The `DiffGenerator` component bridges the gap between algorithmic computation and human-readable output. While the `LCSEngine` identifies which lines are common between files, the diff generator transforms this mathematical result into actionable editing instructions that clearly communicate what changed, where it changed, and how much context surrounds each change.

Mental Model: Editorial Instructions

Think of the `DiffGenerator` as a professional editor creating revision instructions for a manuscript. When an editor compares two drafts of a document, they don't just identify differences—they create clear, structured instructions that tell the author exactly what to add, delete, or keep unchanged. These instructions are grouped into logical sections with enough surrounding context that the author can understand the intent behind each change.

Just as an editor might group related changes together ("In the third paragraph, delete the second sentence and add these two new sentences"), our diff generator groups nearby line changes into **hunks** with surrounding **context lines**. The editor provides context by showing unchanged sentences around the edits, and our diff generator includes unchanged lines around modifications so readers can orient themselves within the file structure.

The editorial analogy extends to the output format: professional editing uses standardized notation (strike-through for deletions, highlights for additions, margin notes for instructions), while our diff generator uses the **unified diff format** with `-` for deletions, `+` for additions, and space-prefixed lines for context. Both systems prioritize clarity and actionability over raw algorithmic output.

Edit Script Generation

The process of converting LCS results into edit operations requires translating from the mathematical representation of common subsequences to the practical representation of file modifications. The LCS algorithm identifies which lines exist in both files, but the diff generator must determine the editing operations needed to transform the first file into the second file.

The conversion algorithm works by walking through both input files simultaneously while consulting the LCS result to determine the fate of each line. This three-way comparison—original file, target file, and common subsequence—allows the generator to categorize every line as unchanged, deleted, or added.

> **Decision: Line-by-Line Edit Operation Classification**
>
> - **Context**: The LCS result identifies common lines but doesn't directly specify edit operations. We need to transform this into explicit ADD, DELETE, and UNCHANGED classifications.
> - **Options Considered**:
>   1. Generate edit operations directly from LCS positions
>   2. Use diff algorithm to reconstruct operations from LCS
>   3. Implement Myers' algorithm for direct edit script generation
> - **Decision**: Generate edit operations by comparing file positions against LCS positions
> - **Rationale**: This approach leverages our existing LCS implementation while remaining conceptually straightforward for learners. Myers' algorithm is more efficient but adds complexity beyond the core learning goals.
> - **Consequences**: Enables clear separation between LCS computation and diff generation, making the code more modular and easier to debug. Performance is adequate for typical file sizes.

The `lcs_to_edit_operations` function implements this classification by maintaining three pointers: one for each input file and one for the LCS positions. As it advances through the files, it determines whether each line should be classified as unchanged (present in LCS), deleted (in first file but not LCS), or added (in second file but not LCS).

Edit Operation	Detection Logic	Line Numbering	Content Source
`UNCHANGED`	Line position matches LCS position	Both old and new line numbers	Content from either file (identical)
`DELETED`	Line in file1 but not at current LCS position	Old line number only	Content from first file
`ADDED`	Line in file2 but not at current LCS position	New line number only	Content from second file

The algorithm maintains careful tracking of line numbers for each file, as the unified diff format requires accurate position information. Deleted lines reference their position in the original file, added lines reference their position in the target file, and unchanged lines reference positions in both files.

Consider a concrete example where the LCS identifies that lines 2, 4, and 7 from the first file correspond to lines 2, 5, and 8 in the second file. The edit script generation proceeds as follows:

1. **Line 1 of file1**: Not in LCS positions, so mark as `DELETED` with `old_line_num=1`
2. **Line 1 of file2**: Not in LCS positions, so mark as `ADDED` with `new_line_num=1`
3. **Line 2 of both files**: Matches LCS position, so mark as `UNCHANGED` with `old_line_num=2, new_line_num=2`
4. **Line 3 of file1**: Not in LCS positions, so mark as `DELETED` with `old_line_num=3`
5. **Lines 3-4 of file2**: Line 3 not in LCS, line 4 not in LCS, so both marked as `ADDED`
6. Continue this pattern through both files

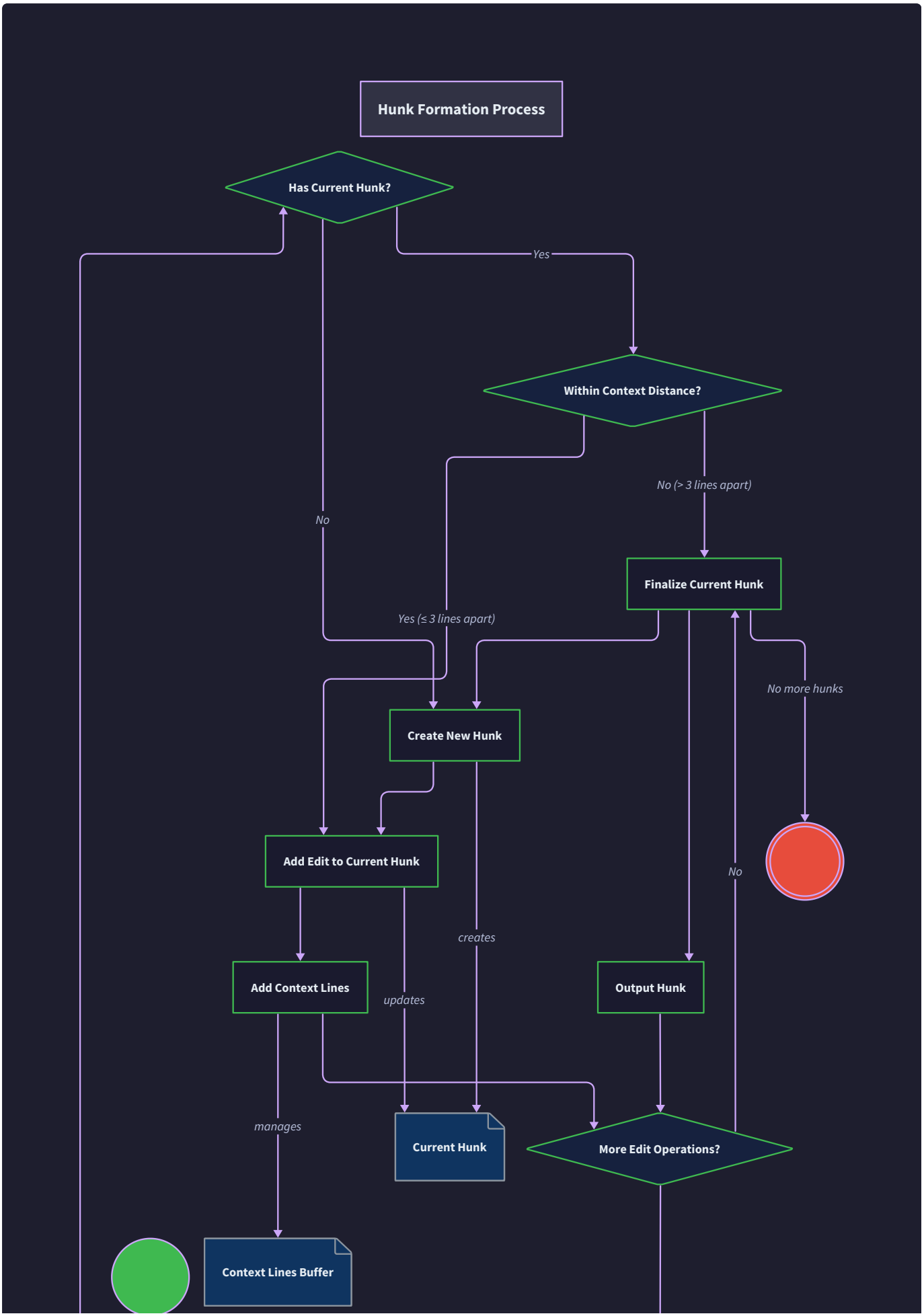
The resulting edit script provides a complete transformation recipe that can convert the first file into the second file through a sequence of line operations.

> The critical insight is that edit operations are not just about identifying changes—they must preserve enough information to reconstruct either file from the other. This is why line numbering accuracy is essential for tools that apply diff output.

Hunk Formation and Context

Raw edit operations provide complete transformation information but lack the contextual grouping that makes diffs readable and actionable. **Hunk formation** addresses this by clustering nearby changes together and surrounding them with unchanged lines that provide visual and logical context.

The hunk formation algorithm operates on the principle that changes separated by only a few unchanged lines are more comprehensible when presented together rather than as separate modification blocks. This mirrors how readers naturally process textual changes—they need enough surrounding context to understand the purpose and scope of modifications.


Hunk Formation Process
Has Current Hunk?
Yes
No
Within Context Distance?
No (> 3 lines apart)
Yes (≤ 3 lines apart)
Finalize Current Hunk
No more hunks
Create New Hunk
Add Edit to Current Hunk
creates
updates
Add Context Lines
Output Hunk
No
manages
Current Hunk
More Edit Operations?
Context Lines Buffer

The `diff_lines_to_hunks` function implements a sliding window approach to group edit operations. It scans through the sequence of `DiffLine` objects, identifying boundaries where hunks should begin and end based on configurable **context line** counts and gap thresholds between changes.

Hunk Formation Parameter	Default Value	Purpose	Impact of Increasing
`context_lines`	3	Lines of unchanged context around each change	More context but larger hunks
`max_gap`	2 × `context_lines`	Maximum unchanged lines between changes in same hunk	Fewer, larger hunks
`min_hunk_size`	1	Minimum number of changed lines per hunk	Filters out tiny modifications

The algorithm maintains state about the current hunk being constructed and decides whether to extend it or start a new one based on the gap between changes. When encountering a sequence of unchanged lines, it determines whether these should become inter-change context (keeping the hunk open) or hunk separator lines (closing the current hunk and starting a new one).

> **Decision: Context Line Strategy**
>
> - **Context**: Diffs need surrounding unchanged lines for readability, but too much context creates overwhelming output while too little context provides insufficient orientation.
> - **Options Considered**:
>   1. Fixed context count (e.g., always 3 lines before and after)
>   2. Adaptive context based on change density
>   3. Configurable context with intelligent merging
> - **Decision**: Configurable fixed context with automatic hunk merging when gaps are small
> - **Rationale**: Provides predictable output size while allowing users to adjust based on their needs. Automatic merging prevents fragmentation of closely-related changes.
> - **Consequences**: Simple implementation that covers most use cases. Users can increase context for complex changes or decrease it for focused reviews.

The hunk boundary detection logic examines sequences of unchanged lines to determine their role:

1. **Leading context**: Up to `context_lines` unchanged lines before the first change in a hunk

2. **Inter-change context**: Unchanged lines between two changes within the same hunk
3. **Trailing context**: Up to `context_lines` unchanged lines after the last change in a hunk
4. **Separator lines**: Unchanged lines that are too numerous to serve as context, indicating a hunk boundary

Each `Hunk` object encapsulates a complete modification unit with its surrounding context:

Hunk Field	Type	Description	Example Value
`old_start`	int	Starting line number in original file (1-indexed)	45
`old_count`	int	Number of lines from original file in this hunk	7
`new_start`	int	Starting line number in target file (1-indexed)	45
`new_count`	int	Number of lines from target file in this hunk	9
`lines`	list[DiffLine]	Complete sequence of context and changed lines	[context, deletion, addition, context]
`context_before`	int	Actual leading context lines included	3
`context_after`	int	Actual trailing context lines included	2

The unified diff format represents each hunk with a header line that summarizes its scope: `@@ -45,7 +45,9 @@`. This header communicates that the hunk starts at line 45 in the original file and includes 7 lines, while starting at line 45 in the target file and including 9 lines. The difference in line counts (9 - 7 = 2) immediately indicates that this hunk represents a net addition of 2 lines.

Context line management requires careful boundary handling to avoid duplicating lines between adjacent hunks or exceeding file boundaries. The algorithm tracks available context at the beginning and end of files, adjusting the actual context counts when insufficient lines exist.

Consider the hunk merging decision process for two potential hunks separated by 4 unchanged lines when `context_lines=3`:

1. **First hunk**: Wants 3 lines of trailing context
2. **Gap**: 4 unchanged lines exist between hunks
3. **Second hunk**: Wants 3 lines of leading context
4. **Total requirement**: 3 + 4 + 3 = 10 lines to represent separately
5. **Merged requirement**: 4 lines as inter-change context
6. **Decision**: Merge hunks since merged representation is more compact

The `merge_hunks` function implements this logic by checking whether the combined representation would be more efficient than separate hunks. This prevents the creation of tiny hunks separated by minimal context, which creates fragmented and hard-to-follow diffs.

Common Pitfalls

Understanding where diff generation commonly goes wrong helps developers avoid frustrating debugging sessions and incorrect output. These pitfalls often stem from the impedance mismatch between zero-indexed programming languages and one-indexed diff formats, as well as edge cases in file structure and algorithm logic.

⚠️ Pitfall: Zero-Based vs One-Based Line Numbering

Programming languages use zero-based indexing for arrays and lists, but the unified diff format uses one-based line numbering for human readability. This creates a persistent source of off-by-one errors that manifest as incorrect `@@ -start,count +start,count @@` headers.

The error typically appears when developers directly use list indices as line numbers in the diff output. For example, if a change occurs at index 5 in a Python list, the correct diff line number is 6, not 5. This mistake becomes particularly problematic when tools attempt to apply the generated diff, as they expect standard one-based numbering.

The fix requires consistent application of `+1` conversion when generating line numbers for `DiffLine` objects and hunk headers. However, the conversion must only apply to external output—internal algorithm logic should continue using zero-based indices for array access.

```
# WRONG: Using zero-based indices directly
hunk.old_start = first_change_index  # Results in line 0, which is invalid

# CORRECT: Converting to one-based for external format
hunk.old_start = first_change_index + 1  # Results in line 1
```

⚠️ Pitfall: Files With No Common Lines

When two files share no common content, the LCS algorithm returns an empty subsequence, which can cause the diff generator to produce degenerate output or crash entirely. This scenario occurs more frequently than expected—when comparing completely different files, generated files with different content, or files where one is empty.

The symptom appears as either a single massive hunk containing all deletions followed by all additions, or as an error when the algorithm assumes at least some common content exists. Some implementations incorrectly try to create context lines when none exist, leading to array access errors.

The correct approach treats files with no common lines as a special case: create a single hunk that deletes all lines from the first file and adds all lines from the second file. No context lines exist by definition, so the hunk contains only change operations.

⚠️ Pitfall: Context Line Overlap Between Adjacent Hunks

When changes occur close together in files, the trailing context of one hunk may overlap with the leading context of the next hunk. Naive implementations either duplicate these lines (creating invalid diffs) or crash when trying to create conflicting line number ranges.

This manifests as hunks with overlapping line number ranges in their headers, such as `@@ -10,8 +10,9 @@` followed immediately by `@@ -15,6 +16,7 @@`. The overlap between lines 15-17 creates ambiguity about which hunk owns those lines.

The solution requires either automatic hunk merging when overlap is detected, or intelligent context truncation that ensures clean boundaries between hunks. The `merge_hunks` function should be called whenever the distance between hunks is less than `2 * context_lines + 1`.

⚠️ Pitfall: Empty File Handling

Empty files create edge cases in multiple parts of the diff generation pipeline. When one or both input files are empty, the LCS algorithm may return unexpected results, line numbering becomes ambiguous, and hunk generation may fail to handle the boundary conditions properly.

The specific failure modes include attempting to access line indices that don't exist, creating hunks with invalid start positions (like line 0), and generating context lines from files that have no content. Some implementations crash when trying to create `DiffLine` objects from non-existent content.

Proper empty file handling requires explicit checks at the beginning of the diff generation process. Empty files should generate straightforward hunks: if the first file is empty, create a single hunk that adds all lines from the second file starting at line 1. If the second file is empty, create a single hunk that deletes all lines from the first file.

⚠️ Pitfall: Line Content Modification vs Line Replacement

The unified diff format represents modified lines as a deletion followed by an addition, not as a single "change" operation. Developers familiar with other diff tools may attempt to create a "MODIFIED" line type, which doesn't exist in the standard format and breaks compatibility with external tools.

This misconception leads to implementations that try to highlight changed portions within lines or create custom diff formats that aren't interoperable with standard tools like `patch`, `git apply`, or text editors that understand unified diff format.

The correct approach always represents line modifications as separate DELETE and ADD operations, even when the lines are nearly identical. Word-level or character-level highlighting is a presentation concern that should be handled by output formatting, not by the core diff generation logic.

Pitfall	Symptom	Root Cause	Fix
Off-by-one line numbers	`patch` command fails to apply diff	Using zero-based indices in output	Add 1 when generating line numbers for external format
No common lines crash	Algorithm error with empty LCS	Assuming LCS contains elements	Check for empty LCS and handle as special case
Overlapping hunk contexts	Invalid diff with conflicting line ranges	Not merging nearby changes	Implement hunk merging when gap < 2*context + 1
Empty file errors	Array access errors or invalid hunks	Not handling missing content	Explicit empty file checks with appropriate hunk generation
Invalid "MODIFIED" operations	Incompatible diff output	Misunderstanding unified diff format	Always use separate DELETE + ADD for changed lines

Implementation Guidance

The `DiffGenerator` component transforms the mathematical output from `LCSEngine` into human-readable editing instructions. This implementation bridges algorithmic computation with practical text processing requirements.

A. Technology Recommendations

Component	Simple Option	Advanced Option
Edit Operation Storage	Python lists with dataclasses	Custom linked list for memory efficiency
Hunk Generation	Linear scan with state machine	Streaming parser for large files
Line Number Tracking	Separate counters for each file	Unified position tracker with offsets
Context Management	Fixed-size windows	Adaptive context based on change density

B. Recommended File Structure

```
diff_tool/
  src/
    diff_generator/
      __init__.py              ← exports DiffGenerator, LineType, Hunk
      diff_generator.py        ← main DiffGenerator class
      edit_operations.py       ← DiffLine, LineType, edit script generation
      hunk_formatter.py        ← Hunk class and formation logic
      context_manager.py       ← context line extraction and merging
    tests/
      test_diff_generator.py   ← comprehensive test suite
      test_hunk_formation.py   ← hunk boundary and merging tests
      fixtures/
        simple_diff/           ← basic test cases
        edge_cases/            ← empty files, no common lines
        large_files/           ← performance test cases
```

C. Infrastructure Starter Code

edit_operations.py - Complete implementation for line classification:

```python
from enum import Enum

from dataclasses import dataclass

from typing import List, Optional, Tuple


class LineType(Enum):

    """Classification of lines in diff output."""

    UNCHANGED = "unchanged"

    ADDED = "added"

    DELETED = "deleted"


@dataclass

class DiffLine:

    """Represents a single line in the diff with its classification and position."""

    content: str

    line_type: LineType

    old_line_num: Optional[int]  # None for ADDED lines

    new_line_num: Optional[int]  # None for DELETED lines



    def is_change(self) -> bool:

        """Returns True if this line represents a modification (ADD or DELETE)."""

        return self.line_type in (LineType.ADDED, LineType.DELETED)



    def format_unified_diff_line(self) -> str:

        """Format this line for unified diff output with appropriate prefix."""

        if self.line_type == LineType.ADDED:

            return f"+{self.content}"

        elif self.line_type == LineType.DELETED:

            return f"-{self.content}"

        else:  # UNCHANGED
```

```
            return f" {self.content}"


def create_diff_line(content: str, line_type: LineType,
                     old_num: Optional[int] = None,
                     new_num: Optional[int] = None) -> DiffLine:
    """Factory function for creating properly validated DiffLine objects."""
    # Validate line number consistency with line type
    if line_type == LineType.ADDED and old_num is not None:
        raise ValueError("ADDED lines should not have old_line_num")
    if line_type == LineType.DELETED and new_num is not None:
        raise ValueError("DELETED lines should not have new_line_num")
    if line_type == LineType.UNCHANGED and (old_num is None or new_num is None):
        raise ValueError("UNCHANGED lines must have both line numbers")


    return DiffLine(content, line_type, old_num, new_num)
```

hunk_formatter.py - Complete implementation for hunk data structure:

PYTHON

```python
from dataclasses import dataclass
from typing import List, Optional
from .edit_operations import DiffLine, LineType


@dataclass
class Hunk:
    """Represents a group of related changes with surrounding context."""
    old_start: int          # Starting line number in original file (1-indexed)
    old_count: int          # Number of lines from original file in this hunk
    new_start: int          # Starting line number in target file (1-indexed)
    new_count: int          # Number of lines from target file in this hunk
    lines: List[DiffLine]   # Complete sequence of context and changed lines
    context_before: int     # Actual leading context lines included
    context_after: int      # Actual trailing context lines included


    def format_hunk_header(self) -> str:
        """Generate the @@ header line for this hunk."""
        return f"@@ -{self.old_start},{self.old_count} +{self.new_start},{self.new_count} @@"


    def format_unified_diff(self) -> List[str]:
        """Generate complete unified diff output for this hunk."""
        result = [self.format_hunk_header()]
        result.extend(line.format_unified_diff_line() for line in self.lines)
        return result


    def change_count(self) -> int:
        """Count the number of actual changes (non-context lines) in this hunk."""
        return sum(1 for line in self.lines if line.is_change())
```

```
def validate_hunk_consistency(hunk: Hunk) -> bool:
    """Verify that hunk line counts match actual content."""
    old_lines = sum(1 for line in hunk.lines
                    if line.line_type in (LineType.UNCHANGED, LineType.DELETED))
    new_lines = sum(1 for line in hunk.lines
                    if line.line_type in (LineType.UNCHANGED, LineType.ADDED))

    return (old_lines == hunk.old_count and
            new_lines == hunk.new_count)

def merge_hunks(hunk1: Hunk, hunk2: Hunk, max_gap: int) -> Optional[Hunk]:
    """Merge two adjacent hunks if the gap between them is small enough."""
    # Calculate gap between hunks
    hunk1_end = hunk1.old_start + hunk1.old_count - 1
    gap_size = hunk2.old_start - hunk1_end - 1

    if gap_size > max_gap:
        return None  # Gap too large to merge

    # Create merged hunk
    merged_old_start = hunk1.old_start
    merged_old_count = hunk2.old_start + hunk2.old_count - hunk1.old_start
    merged_new_start = hunk1.new_start
    merged_new_count = hunk2.new_start + hunk2.new_count - hunk1.new_start

    # Combine line sequences (implementation would need gap lines added)
    merged_lines = hunk1.lines + hunk2.lines  # Simplified - needs gap handling
```

```
    return Hunk(
        old_start=merged_old_start,
        old_count=merged_old_count,
        new_start=merged_new_start,
        new_count=merged_new_count,
        lines=merged_lines,
        context_before=hunk1.context_before,
        context_after=hunk2.context_after
    )
```

D. Core Logic Skeleton Code

diff_generator.py - Main component for learners to implement:

PYTHON

```python
from typing import List, Tuple
from .edit_operations import DiffLine, LineType, create_diff_line
from .hunk_formatter import Hunk, validate_hunk_consistency, merge_hunks

class DiffGenerator:
    """Converts LCS results into edit operations and groups them into hunks."""

    def __init__(self, default_context_lines: int = 3):
        self.default_context_lines = default_context_lines

    def lcs_to_edit_operations(self, lcs: 'CommonSubsequence',
                               file1_lines: List[str],
                               file2_lines: List[str]) -> List[DiffLine]:
        """Convert LCS result to sequence of edit operations.

        Args:
            lcs: CommonSubsequence result from LCSEngine
            file1_lines: Lines from original file
            file2_lines: Lines from target file

        Returns:
            Complete sequence of DiffLine objects representing the transformation
        """
        # TODO 1: Handle empty LCS case (no common lines between files)
        # TODO 2: Initialize pointers for file1, file2, and LCS positions
        # TODO 3: Create main loop that advances through both files simultaneously
        # TODO 4: For each position, check if current lines match LCS positions
        # TODO 5: If lines match LCS, create UNCHANGED DiffLine with both line numbers
```

```
        # TODO 6: If file1 line doesn't match LCS, create DELETED DiffLine
        # TODO 7: If file2 line doesn't match LCS, create ADDED DiffLine
        # TODO 8: Advance appropriate pointers based on operation type
        # TODO 9: Handle remaining lines when one file is exhausted
        # TODO 10: Return complete sequence of DiffLine objects

        # Hint: Use enumerate() to track line numbers while iterating
        # Hint: LCS positions list tells you which lines are common
        # Hint: Convert 0-based list indices to 1-based line numbers for output
        pass

    def diff_lines_to_hunks(self, diff_lines: List[DiffLine],
                            context_lines: int = None) -> List[Hunk]:
        """Group edit operations into hunks with surrounding context.

        Args:
            diff_lines: Sequence of DiffLine objects from lcs_to_edit_operations
            context_lines: Number of context lines around changes (uses default if None)

        Returns:
            List of Hunk objects with changes grouped and context included
        """
        if context_lines is None:
            context_lines = self.default_context_lines

        # TODO 1: Handle empty diff_lines list (identical files)
        # TODO 2: Scan through diff_lines to identify change boundaries
        # TODO 3: For each change group, collect surrounding context lines
```

```
        # TODO 4: Determine hunk start/end positions in both files
        # TODO 5: Extract context_before lines (up to context_lines)
        # TODO 6: Include all change lines in the group
        # TODO 7: Extract context_after lines (up to context_lines)
        # TODO 8: Create Hunk object with proper line counts and positions
        # TODO 9: Validate hunk consistency using validate_hunk_consistency()
        # TODO 10: Check if adjacent hunks should be merged using merge_hunks()
        # TODO 11: Return final list of non-overlapping hunks

        # Hint: Track current position in both original and target files
        # Hint: Use sliding window to collect context around changes
        # Hint: max_gap for merging should be 2 * context_lines
        pass

    def generate_diff(self, file1_lines: List[str], file2_lines: List[str],
                      lcs: 'CommonSubsequence', context_lines: int = None) -> List[Hunk]:
        """Main entry point: convert LCS result to structured diff hunks.

        Args:
            file1_lines: Lines from original file
            file2_lines: Lines from target file
            lcs: CommonSubsequence result from LCSEngine.compute_lcs()
            context_lines: Number of context lines around changes

        Returns:
            List of Hunk objects ready for formatting and output
        """
        # TODO 1: Call lcs_to_edit_operations() to get edit sequence
```

```
        # TODO 2: Call diff_lines_to_hunks() to group operations with context
        # TODO 3: Return structured hunks ready for output formatting


        # This is the main pipeline function that coordinates the conversion
        pass
```

E. Language-Specific Hints

- **Line Number Handling**: Python lists are 0-indexed but diff format uses 1-indexed line numbers. Always add 1 when creating external output.
- **Dataclass Usage**: Use `@dataclass` for `DiffLine` and `Hunk` to get automatic `__init__`, `__repr__`, and comparison methods.
- **Enum for LineType**: Use `enum.Enum` for `LineType` to ensure type safety and prevent invalid line classifications.
- **List Slicing**: Use `lines[start:end]` for extracting context windows around changes.
- **Optional Type Hints**: Use `Optional[int]` for line numbers that may be None (deleted lines don't have new numbers, added lines don't have old numbers).
- **Generator Functions**: Consider using `yield` for large files to avoid loading entire diff in memory.

F. Milestone Checkpoint

After implementing the `DiffGenerator` component, verify correct behavior:

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

Expected Output: All tests pass, including edge cases for empty files, no common lines, and context merging.

Manual Verification:

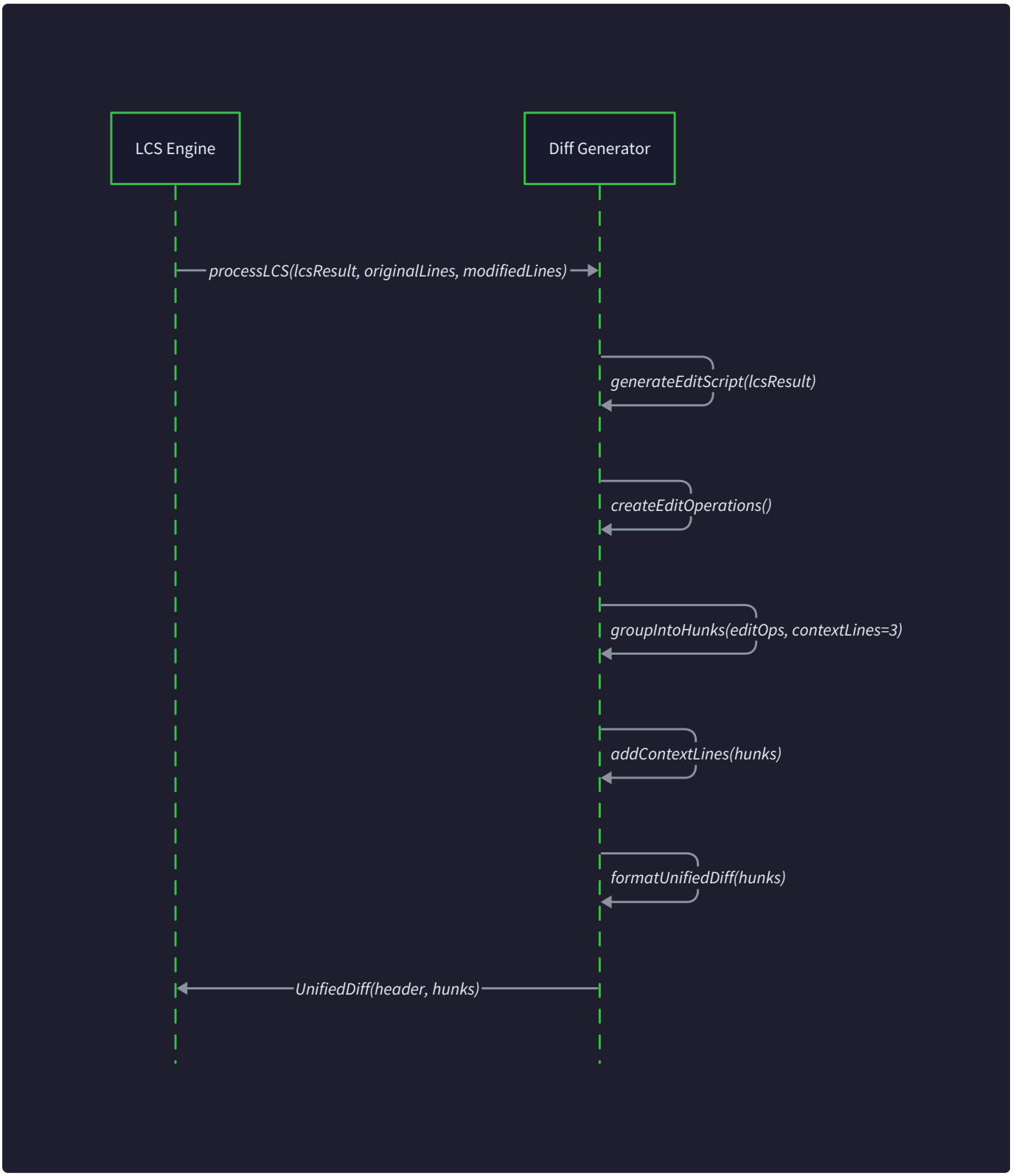
```bash
# Test with simple two-line change
python diff_tool.py file1.txt file2.txt

# Expected output format:
# @@ -1,4 +1,5 @@
#  unchanged line
# -deleted line
# +added line
# +another added line
#  unchanged line
```

Signs of Problems:

- Line numbers start at 0 instead of 1 → Check line number conversion
- Context lines missing → Check hunk boundary calculation
- Overlapping hunks → Implement hunk merging logic
- Crash on empty files → Add empty file handling

G. Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
`patch` command rejects diff	Incorrect line numbering	Check hunk headers for 0-based numbers	Add 1 to all line numbers in output
Missing context lines	Wrong boundary calculation	Print hunk start/end positions	Fix context extraction window logic
Duplicate lines in output	Overlapping hunks not merged	Check gap calculation between hunks	Implement merge_hunks() properly
Crash on identical files	Empty diff_lines list not handled	Test with identical input files	Add empty list check in diff_lines_to_hunks
Wrong hunk line counts	Inconsistent counting logic	Use validate_hunk_consistency()	Count UNCHANGED+DELETED for old, UNCHANGED+ADDED for new



The `DiffGenerator` component completes the transformation from mathematical algorithm output to practical editing instructions. When properly implemented, it provides the foundation for human-readable diffs that clearly communicate file changes while maintaining compatibility with standard diff tools and workflows.

Output Formatter Component

> **Milestone(s):** Milestone 4: CLI and Color Output — builds command-line interface with colored output and options, plus foundational unified diff format from Milestone 3

The `OutputFormatter` component represents the final stage in our diff processing pipeline, transforming structured `Hunk` objects into human-readable unified diff format with optional color highlighting and command-line interface integration. This component bridges the gap between our internal diff representation and the standardized output formats that users expect from professional diff tools.

Mental Model: Publishing Changes

Think of the `OutputFormatter` as a publishing house editor preparing editorial changes for publication. When an author submits a revised manuscript, the editor doesn't just show the raw list of insertions and deletions. Instead, they format the changes using standardized editorial notation — strikethroughs for deletions, highlighting for additions, section markers to show where changes occur, and surrounding context so readers can understand the modifications in their proper setting.

Similarly, our formatter takes the structured diff information from the `DiffGenerator` and presents it using the universally recognized unified diff format. Just as a publisher might produce different versions of the same content — a black-and-white edition for print, a colorized version for digital readers, or a plain text version for accessibility — our formatter adapts its output based on the target audience and display capabilities.

The unified diff format serves as the "standard editorial notation" of the software world. Every developer recognizes the `-` and `+` prefixes, the `@@` hunk markers, and the context lines surrounding changes. Our job is to generate this format correctly while enhancing it with modern features like color highlighting when appropriate.

Unified Diff Format

The unified diff format represents the lingua franca of change visualization in software development. This standardized format emerged from the Unix `diff` utility and has been adopted across virtually every version control system, code review tool, and text comparison application. Understanding its structure is crucial for generating output that integrates seamlessly with existing development workflows.

Format Structure and Components

The unified diff format consists of several distinct sections, each serving a specific purpose in communicating changes between files. At the highest level, a unified diff contains a header section identifying the files being compared, followed by one or more hunks representing groups of changes.

Component	Format	Purpose	Example
File Headers	`--- filename1` and `+++ filename2`	Identify source and target files	`--- file1.txt`
Hunk Header	`@@ -old_start,old_count +new_start,new_count @@`	Define line ranges for changes	`@@ -15,7 +15,9 @@`
Context Lines	`content` (space prefix)	Show unchanged lines around changes	`def calculate_sum():`
Deleted Lines	`-content` (minus prefix)	Show lines removed from original	`- return x + y`
Added Lines	`+content` (plus prefix)	Show lines added in new version	`+ return x + y + z`
Optional Context	Additional metadata after `@@`	Provide semantic context	`@@ -15,7 +15,9 @@ class Calculator:`

The mathematical precision of the hunk header format deserves special attention. The `@@ -old_start,old_count +new_start,new_count @@` syntax encodes exactly which lines from each file are represented in the hunk. The `old_start` indicates the line number where the hunk begins in the original file, while `old_count` specifies how many lines from the original file are included (both context and deleted lines). Similarly, `new_start` and `new_count` define the corresponding range in the modified file.

> **Critical Insight**: The unified diff format uses **one-indexed line numbers** for human readability, while most programming languages use zero-indexed arrays internally. This off-by-one translation is a frequent source of bugs in diff implementations.

Hunk Formation and Context Integration

The process of converting our internal `Hunk` structures into unified diff format requires careful attention to line numbering, prefix selection, and context integration. Each `DiffLine` within a hunk must be formatted according to its `LineType`, with appropriate prefixes and line number tracking.

```
Format Generation Algorithm:
1. Generate file headers using original file paths
2. For each hunk in the diff:
   a. Calculate actual line ranges including context
   b. Generate hunk header with @@ syntax
   c. Format each DiffLine with appropriate prefix
   d. Track line numbers for both files separately
   e. Ensure context lines appear in correct positions
3. Validate that line counts match hunk headers
```

The line number calculation requires maintaining separate counters for the original and modified files as we process each hunk. Context lines contribute to both file's line counts, deleted lines only increment the original file counter, and added lines only increment the modified file counter.

> **Decision: Line Number Tracking Strategy**
>
> - **Context**: Need to generate accurate hunk headers with correct line ranges
> - **Options Considered**:
>   1. Pre-calculate all line numbers before formatting
>   2. Track incrementally during formatting
>   3. Store line numbers in DiffLine structures
> - **Decision**: Track incrementally during formatting with validation
> - **Rationale**: Incremental tracking is memory-efficient and naturally handles edge cases like empty files or hunks with only additions/deletions
> - **Consequences**: Requires careful counter management but provides flexibility for different output formats

Line Type	Original Counter	Modified Counter	Output Prefix
UNCHANGED	+1	+1	` ` (space)
DELETED	+1	+0	`-`
ADDED	+0	+1	`+`

Header Generation and File Identification

The unified diff header serves a crucial role in identifying the files being compared and providing metadata for tools that consume the diff output. The standard format uses `---` to identify the original file and `+++` to identify the modified file, followed by optional timestamps and file mode information.

Our implementation focuses on the essential file identification while remaining compatible with standard diff consumers. The header generation must handle several edge cases, including missing files (for new file creation or deletion scenarios), different file paths, and special cases like comparing content from standard input.

Header Type	Format	Usage	Example
Original File	`--- filepath`	Identifies source file	`--- /path/to/original.txt`
Modified File	`+++ filepath`	Identifies target file	`+++ /path/to/modified.txt`
New File	`--- /dev/null`	Indicates file creation	When comparing against empty
Deleted File	`+++ /dev/null`	Indicates file deletion	When file was removed
Stdin Input	`--- -` or `+++ -`	Standard input source	For pipe input scenarios

Color Output and CLI Interface

Modern terminal applications enhance user experience through visual highlighting and interactive features. Our color output system must intelligently detect terminal capabilities, provide user control over formatting, and gracefully degrade when colors are inappropriate or unsupported.

ANSI Color Support and TTY Detection

Color output relies on ANSI escape sequences to control text formatting in compatible terminals. However, these control codes can corrupt output when redirected to files or consumed by tools that don't understand ANSI formatting. Our implementation must detect the output destination and apply colors appropriately.

ANSI Code	Constant	Purpose	Terminal Effect
`\033[31m`	`RED`	Highlight deletions	Red text
`\033[32m`	`GREEN`	Highlight additions	Green text
`\033[0m`	`RESET`	Reset formatting	Return to normal
`\033[1m`	`BOLD`	Emphasize headers	Bold text
`\033[2m`	`DIM`	De-emphasize context	Dimmed text

The TTY detection process determines whether the output destination supports color formatting. This involves checking if the output is connected to a terminal device rather than being redirected to a file or piped to another program.

```
TTY Detection Algorithm:
1. Check if output destination is a terminal device
2. Verify TERM environment variable indicates color support
3. Respect user preferences via --no-color flag
4. Default to plain text for non-interactive environments
5. Provide --force-color flag for special cases
```

Decision: Color Application Strategy

- **Context**: Need to provide helpful visual highlighting while maintaining compatibility with scripts and tools
- **Options Considered**:
  1. Always apply colors and let users disable with flags
  2. Auto-detect TTY and apply colors only when appropriate
  3. Default to no colors and require explicit enabling
- **Decision**: Auto-detect TTY with explicit override flags
- **Rationale**: Provides the best user experience for interactive use while being safe for scripting and tool integration
- **Consequences**: Requires platform-specific TTY detection but maximizes compatibility

Command-Line Argument Processing

The CLI interface provides users with control over diff formatting, output options, and comparison behavior. The argument parsing system must handle file path validation, option conflicts, and provide helpful error messages for invalid usage.

Argument	Type	Purpose	Default	Validation
`file1`	positional	First file to compare	required	Must exist and be readable
`file2`	positional	Second file to compare	required	Must exist and be readable
`--context N`	optional	Context lines around changes	3	Must be non-negative integer
`--no-color`	flag	Disable ANSI color output	false	Conflicts with --force-color
`--force-color`	flag	Enable color even for non-TTY	false	Conflicts with --no-color
`--side-by-side`	flag	Enable parallel display mode	false	Future extension
`--unified N`	optional	Unified format context lines	3	Synonym for --context

The argument validation process must handle various error conditions gracefully, providing specific guidance for common mistakes. File path validation should distinguish between non-existent files, permission errors, and binary file detection.

```
Argument Processing Flow:
1. Parse command-line arguments using standard library
2. Validate file paths and check accessibility
3. Resolve option conflicts with clear error messages
4. Set up formatting preferences based on TTY detection
5. Initialize OutputFormatter with validated configuration
```

Exit Code Conventions

Following Unix conventions, our diff tool communicates results through exit codes that scripts and automation tools can interpret programmatically. The exit code convention allows other programs to determine the comparison result without parsing output text.

Exit Code	Condition	Meaning	Script Usage
0	Files identical	No differences found	Success in automation
1	Files differ	Differences detected	Expected in many workflows
2	Error occurred	File access, parsing, or system error	Requires intervention

The exit code generation must occur after all processing completes successfully. Even if differences are found, the program should exit with code 1 (not an error condition) only after generating complete diff output.

ColorFormatter Helper Implementation

The `ColorFormatter` component encapsulates ANSI color logic and provides a clean interface for applying formatting to diff output. This separation allows the main formatting logic to remain readable while isolating platform-specific color handling.

Method	Parameters	Returns	Purpose
`red(text)`	text: str	str	Apply red formatting to text
`green(text)`	text: str	str	Apply green formatting to text
`bold(text)`	text: str	str	Apply bold formatting to text
`reset()`	none	str	Return RESET escape code
`is_color_enabled()`	none	bool	Check if colors should be applied
`format_diff_line(line, line_type)`	line: str, line_type: LineType	str	Apply appropriate color for line type

The color formatting methods should be designed to be no-ops when color output is disabled, allowing the main formatting code to call them unconditionally without performance concerns.

Common Pitfalls

Understanding the common mistakes in diff output formatting helps avoid compatibility issues and user experience problems that can make an otherwise correct diff tool frustrating to use.

⚠️ Pitfall: ANSI Codes in Redirected Output

One of the most frequent issues occurs when ANSI color codes are written to files or pipes, corrupting the output for tools that expect plain text. This happens when developers test their diff tool in a terminal (where colors work correctly) but don't verify behavior when output is redirected.

```
# Problem: ANSI codes corrupt piped output
./diff file1.txt file2.txt | less
# Shows: ^[[32m+added line^[[0m instead of: +added line

# Problem: ANSI codes in saved files
./diff file1.txt file2.txt > changes.diff
# File contains control characters unusable by other tools
```

The root cause is failing to detect that output is being redirected rather than displayed in a terminal. The solution requires proper TTY detection that checks whether standard output is connected to a terminal device.

Detection Strategy: Check if `sys.stdout.isatty()` returns `True` in Python, or equivalent platform-specific calls. Only enable colors when output goes to an actual terminal.

Recovery Approach: Provide `--no-color` and `--force-color` flags so users can override automatic detection when needed for special cases.

⚠️ Pitfall: Windows Console Color Handling

Windows terminals historically required special initialization to enable ANSI color support, leading to diff tools that work correctly on Unix systems but display raw escape codes on Windows. Modern Windows terminals support ANSI codes, but older systems and certain console applications may not.

```
# Problem on older Windows: Raw escape codes displayed
C:\> diff file1.txt file2.txt
←[31m-deleted line←[0m  # Shows literal escape characters
```

Detection Strategy: On Windows, attempt to enable ANSI support using platform-specific APIs, and fall back to plain text if unsuccessful.

Compatibility Approach: Consider providing Windows-specific color APIs as an alternative to ANSI codes for maximum compatibility.

⚠️ Pitfall: Incorrect Line Number Indexing

The unified diff format uses one-indexed line numbers for human readability, but internal processing typically uses zero-indexed arrays. This mismatch creates off-by-one errors that make the diff output incompatible with other tools.

```
# Problem: Zero-indexed line numbers in output
@@ -0,5 +0,7 @@  # Should be @@ -1,5 +1,7 @@
 context line
-deleted line
+added line
```

Root Cause: Directly using internal array indices in hunk headers without converting to one-indexed format.

Solution Pattern: Maintain separate counters for display line numbers that start at 1, while keeping internal processing zero-indexed.

⚠️ Pitfall: Inconsistent Exit Code Usage

Scripts and automation tools rely on exit codes to determine diff results programmatically. Using non-standard exit codes or returning error codes when files simply differ breaks integration with existing workflows.

```
# Problem: Wrong exit code for differences
$ diff file1.txt file2.txt; echo $?
Files differ
2  # Should be 1, not 2 (2 indicates error, not differences)
```

Standard Convention: Exit 0 for identical files, exit 1 for different files, exit 2 for errors (file not found, permission denied, etc.).

Implementation Pattern: Distinguish between "differences found" (expected outcome, exit 1) and "processing failed" (unexpected error, exit 2).

⚠️ Pitfall: Context Line Count Mismatches

The hunk header declares how many lines from each file are included in the hunk, but counting errors in the implementation can cause the declared counts to mismatch the actual line content, breaking tools that parse diff output.

```
# Problem: Header claims 5 lines but content shows 6
@@ -10,5 +10,5 @@
 context1
 context2
-deleted line
+added line
 context3
 context4  # 6th line but header claims 5
```

Root Cause: Forgetting to include context lines in the count, or miscounting when changes affect the total line count.

Validation Strategy: After generating each hunk, verify that the actual line count matches the declared count in the header.

⚠️ Pitfall: Missing File Header Information

Some diff output omits the required `---` and `+++` file headers, making the output incompatible with patch tools and version control systems that expect standard unified diff format.

```
# Problem: Missing file identification headers
@@ -1,3 +1,3 @@  # Where are the --- and +++ headers?
-old line
+new line
 context
```

Standard Requirement: Every unified diff must begin with `--- original_file` and `+++ modified_file` headers before any hunk content.

Implementation Pattern: Always generate file headers as the first output, even for simple file comparisons.

Implementation Guidance

The `OutputFormatter` transforms structured diff data into human-readable unified diff format with optional color enhancement and CLI integration. This component requires careful attention to formatting standards, terminal compatibility, and command-line conventions.

Technology Recommendations

Component	Simple Option	Advanced Option
Color Support	Basic ANSI codes with TTY detection	Rich color library with theme support
CLI Parsing	argparse standard library	Click framework with subcommands
Text Formatting	String formatting with templates	Dedicated template engine
Terminal Detection	sys.stdout.isatty() check	termcolor library with capability detection

Recommended File Structure

```
project-root/
├── src/
│   ├── diff_tool/
│   │   ├── formatters/
│   │   │   ├── __init__.py
│   │   │   ├── output_formatter.py     ← Main OutputFormatter class
│   │   │   ├── color_formatter.py      ← ANSI color helper
│   │   │   └── unified_diff.py         ← Unified diff format logic
│   │   ├── cli/
│   │   │   ├── __init__.py
│   │   │   ├── argument_parser.py      ← CLI argument processing
│   │   │   └── main.py                 ← Entry point with exit codes
│   │   └── models/
│   │       └── diff_types.py           ← Hunk, DiffLine types
├── tests/
│   ├── formatters/
│   │   ├── test_output_formatter.py
│   │   └── test_color_formatter.py
└── main.py                             ← Application entry point
```

Infrastructure Starter Code

Color Formatter Helper (`src/diff_tool/formatters/color_formatter.py`):

```python
import sys

import os

from typing import Optional

from enum import Enum


class ColorMode(Enum):

    AUTO = "auto"

    ALWAYS = "always"

    NEVER = "never"


class ColorFormatter:

    # ANSI color constants

    RED = "\033[31m"

    GREEN = "\033[32m"

    BOLD = "\033[1m"

    DIM = "\033[2m"

    RESET = "\033[0m"



    def __init__(self, color_mode: ColorMode = ColorMode.AUTO):

        self.color_mode = color_mode

        self._color_enabled = self._determine_color_support()



    def _determine_color_support(self) -> bool:

        """Determine if color output should be enabled based on mode and terminal."""

        if self.color_mode == ColorMode.NEVER:

            return False

        elif self.color_mode == ColorMode.ALWAYS:

            return True

        else:  # AUTO mode
```

```
            return (
                sys.stdout.isatty() and
                os.getenv('TERM', 'dumb') != 'dumb' and
                os.getenv('NO_COLOR') is None
            )

    def is_color_enabled(self) -> bool:
        return self._color_enabled

    def red(self, text: str) -> str:
        if self._color_enabled:
            return f"{self.RED}{text}{self.RESET}"
        return text

    def green(self, text: str) -> str:
        if self._color_enabled:
            return f"{self.GREEN}{text}{self.RESET}"
        return text

    def bold(self, text: str) -> str:
        if self._color_enabled:
            return f"{self.BOLD}{text}{self.RESET}"
        return text

    def dim(self, text: str) -> str:
        if self._color_enabled:
            return f"{self.DIM}{text}{self.RESET}"
```

```
        return text
```

CLI Argument Parser (`src/diff_tool/cli/argument_parser.py`):

```python
import argparse

import sys

from pathlib import Path

from typing import Tuple, Optional

from ..formatters.color_formatter import ColorMode


class DiffArguments:

    def __init__(self, file1: Path, file2: Path, context_lines: int = 3,

                 color_mode: ColorMode = ColorMode.AUTO):

        self.file1 = file1

        self.file2 = file2

        self.context_lines = context_lines

        self.color_mode = color_mode


def parse_arguments() -> DiffArguments:

    parser = argparse.ArgumentParser(

        description="Compare two text files and show differences",

        formatter_class=argparse.RawDescriptionHelpFormatter,

        epilog="""

Exit codes:

  0 - files are identical

  1 - files differ

  2 - error occurred

        """

    )



    parser.add_argument('file1', type=Path, help='First file to compare')

    parser.add_argument('file2', type=Path, help='Second file to compare')
```

```
    parser.add_argument('-c', '--context', type=int, default=3,
                        help='Number of context lines around changes (default: 3)')
    parser.add_argument('-u', '--unified', type=int, dest='context',
                        help='Unified format with N context lines (synonym for --context)')

    color_group = parser.add_mutually_exclusive_group()
    color_group.add_argument('--color', action='store_const',
                             dest='color_mode', const=ColorMode.ALWAYS,
                             help='Force color output even when not a TTY')
    color_group.add_argument('--no-color', action='store_const',
                             dest='color_mode', const=ColorMode.NEVER,
                             help='Disable color output')

    parser.set_defaults(color_mode=ColorMode.AUTO)

    args = parser.parse_args()

    # Validate files exist and are readable
    for file_path in [args.file1, args.file2]:
        if not file_path.exists():
            print(f"Error: File '{file_path}' not found", file=sys.stderr)
            sys.exit(2)
        if not file_path.is_file():
            print(f"Error: '{file_path}' is not a regular file", file=sys.stderr)
            sys.exit(2)

    # Validate context lines
    if args.context < 0:
```

```
        print("Error: Context lines must be non-negative", file=sys.stderr)
        sys.exit(2)


    return DiffArguments(args.file1, args.file2, args.context, args.color_mode)
```

Core Logic Skeleton Code

Output Formatter Main Class (`src/diff_tool/formatters/output_formatter.py`):

```python
from typing import List, TextIO

import sys

from ..models.diff_types import Hunk, DiffLine, LineType, FileContent

from .color_formatter import ColorFormatter, ColorMode


class OutputFormatter:

    def __init__(self, color_mode: ColorMode = ColorMode.AUTO):

        self.color_formatter = ColorFormatter(color_mode)



    def format_unified_diff(self, file1: FileContent, file2: FileContent,

                            hunks: List[Hunk], output: TextIO = sys.stdout) -> None:

        """Generate complete unified diff output with headers and hunks."""

        # TODO 1: Generate and write file headers (--- and +++ lines)

        # TODO 2: Iterate through hunks and format each one

        # TODO 3: For each hunk, write hunk header then format all diff lines

        # TODO 4: Handle empty hunks list (files are identical)

        # Hint: Use _format_file_headers() and _format_hunk() helper methods

        pass



    def _format_file_headers(self, file1: FileContent, file2: FileContent) -> List[str]:

        """Generate --- and +++ header lines identifying the files."""

        # TODO 1: Create --- header with file1 path

        # TODO 2: Create +++ header with file2 path

        # TODO 3: Handle special cases like /dev/null for new/deleted files

        # TODO 4: Return list of header lines

        # Hint: Standard format is "--- filepath" and "+++ filepath"

        pass
```

```
    def _format_hunk(self, hunk: Hunk) -> List[str]:
        """Format a single hunk with header and diff lines."""
        lines = []

        # TODO 1: Generate hunk header with @@ -old_start,old_count +new_start,new_count @@
        # TODO 2: Iterate through hunk.lines and format each DiffLine
        # TODO 3: Apply appropriate prefix (space, -, +) based on line_type
        # TODO 4: Apply color formatting if enabled
        # TODO 5: Validate that actual line count matches header declaration
        # Hint: Use _format_hunk_header() and _format_diff_line() helpers

        return lines

    def _format_hunk_header(self, hunk: Hunk) -> str:
        """Generate the @@ header line for a hunk."""
        # TODO 1: Extract old_start, old_count, new_start, new_count from hunk
        # TODO 2: Format as "@@ -old_start,old_count +new_start,new_count @@"
        # TODO 3: Handle special case where count is 1 (can omit ,1)
        # TODO 4: Apply bold formatting if colors are enabled
        # Hint: Standard unified diff format specification
        pass

    def _format_diff_line(self, diff_line: DiffLine) -> str:
        """Format a single diff line with appropriate prefix and color."""
        # TODO 1: Determine prefix character based on diff_line.line_type
        #         UNCHANGED -> ' ', DELETED -> '-', ADDED -> '+'
        # TODO 2: Combine prefix with diff_line.content
        # TODO 3: Apply color formatting based on line type
```

```
        #         DELETED -> red, ADDED -> green, UNCHANGED -> no color
        # TODO 4: Handle empty lines and lines with only whitespace
        # Hint: Use self.color_formatter.red/green methods
        pass


    def _calculate_line_counts(self, diff_lines: List[DiffLine]) -> tuple[int, int]:
        """Calculate old and new line counts for hunk header."""
        # TODO 1: Count lines that appear in old file (UNCHANGED + DELETED)
        # TODO 2: Count lines that appear in new file (UNCHANGED + ADDED)
        # TODO 3: Return (old_count, new_count) tuple
        # Hint: Context lines count toward both files
        pass
```

Unified Diff Format Logic (`src/diff_tool/formatters/unified_diff.py`):

PYTHON

```python
from typing import List, Optional
from ..models.diff_types import Hunk, DiffLine, LineType


def validate_hunk_consistency(hunk: Hunk) -> bool:
    """Verify that hunk line counts match actual content."""
    # TODO 1: Count actual UNCHANGED and DELETED lines in hunk.lines
    # TODO 2: Count actual UNCHANGED and ADDED lines in hunk.lines
    # TODO 3: Compare with hunk.old_count and hunk.new_count
    # TODO 4: Return True only if counts match exactly
    # Hint: This catches common off-by-one errors in hunk generation
    pass

def merge_adjacent_hunks(hunk1: Hunk, hunk2: Hunk, max_gap: int = 3) -> Optional[Hunk]:
    """Merge two hunks if they are close enough together."""
    # TODO 1: Calculate gap between end of hunk1 and start of hunk2
    # TODO 2: If gap <= max_gap, create merged hunk
    # TODO 3: Combine line lists and recalculate ranges
    # TODO 4: Return merged hunk or None if too far apart
    # Hint: This improves readability by reducing fragmented hunks
    pass

def format_line_prefix(line_type: LineType) -> str:
    """Get the standard unified diff prefix for a line type."""
    # TODO 1: Return ' ' for UNCHANGED lines
    # TODO 2: Return '-' for DELETED lines
    # TODO 3: Return '+' for ADDED lines
    # TODO 4: Raise exception for invalid line types
    # Hint: These prefixes are defined by unified diff standard
    pass
```

Language-Specific Hints

Python-Specific Implementation Tips:

- Use `sys.stdout.isatty()` for TTY detection on Unix systems
- Handle Windows console with `colorama` library if advanced color support needed
- Use `argparse` for CLI parsing - it handles help text and error messages automatically
- File encoding detection can use `chardet` library for robust encoding detection
- Use `pathlib.Path` for cross-platform file path handling

Exit Code Management:

PYTHON

```python
# In main.py entry point

import sys

from diff_tool.cli.argument_parser import parse_arguments

from diff_tool.core.diff_engine import DiffEngine


def main():

    try:

        args = parse_arguments()

        engine = DiffEngine()

        hunks = engine.compare_files(args.file1, args.file2)



        formatter = OutputFormatter(args.color_mode)

        formatter.format_unified_diff(file1, file2, hunks)



        # Exit 0 if no differences, 1 if differences found

        exit_code = 0 if not hunks else 1

        sys.exit(exit_code)



    except FileNotFoundError as e:

        print(f"Error: {e}", file=sys.stderr)

        sys.exit(2)

    except PermissionError as e:

        print(f"Error: {e}", file=sys.stderr)

        sys.exit(2)

    except Exception as e:

        print(f"Unexpected error: {e}", file=sys.stderr)

        sys.exit(2)
```

Milestone Checkpoint

After implementing the OutputFormatter component, verify correct behavior with these tests:

Basic Functionality Test:

```bash
# Create test files

echo -e "line1\nline2\nline3" > test1.txt

echo -e "line1\nmodified\nline3" > test2.txt


# Run your diff tool

python -m diff_tool test1.txt test2.txt
```

Expected Output:

```
--- test1.txt
+++ test2.txt
@@ -1,3 +1,3 @@
 line1
-line2
+modified
 line3
```

Color Output Test:

```bash
# Should show colors in terminal

python -m diff_tool test1.txt test2.txt


# Should show no colors when redirected

python -m diff_tool test1.txt test2.txt > output.diff

cat output.diff  # Should contain no ANSI codes
```

Exit Code Verification:

```bash
# Test identical files
cp test1.txt test1_copy.txt
python -m diff_tool test1.txt test1_copy.txt
echo $?  # Should print 0

# Test different files
python -m diff_tool test1.txt test2.txt
echo $?  # Should print 1

# Test missing file
python -m diff_tool test1.txt nonexistent.txt
echo $?  # Should print 2
```

Context Lines Test:

```bash
# Create larger test file
seq 1 20 > large1.txt
seq 1 20 | sed 's/10/modified/' > large2.txt

# Test different context amounts
python -m diff_tool --context 1 large1.txt large2.txt
python -m diff_tool --context 5 large1.txt large2.txt
```

Signs that implementation is working correctly:

- Output matches standard `diff -u` format exactly
- Colors appear in terminal but not in redirected output
- Exit codes follow Unix conventions (0=same, 1=different, 2=error)
- Context lines appear symmetrically around changes
- Line numbers in @@ headers are one-indexed and accurate

Common issues to debug:

- Zero-indexed line numbers in output (should be one-indexed)
- ANSI codes in redirected output (TTY detection failed)
- Wrong exit codes (check error vs difference handling)

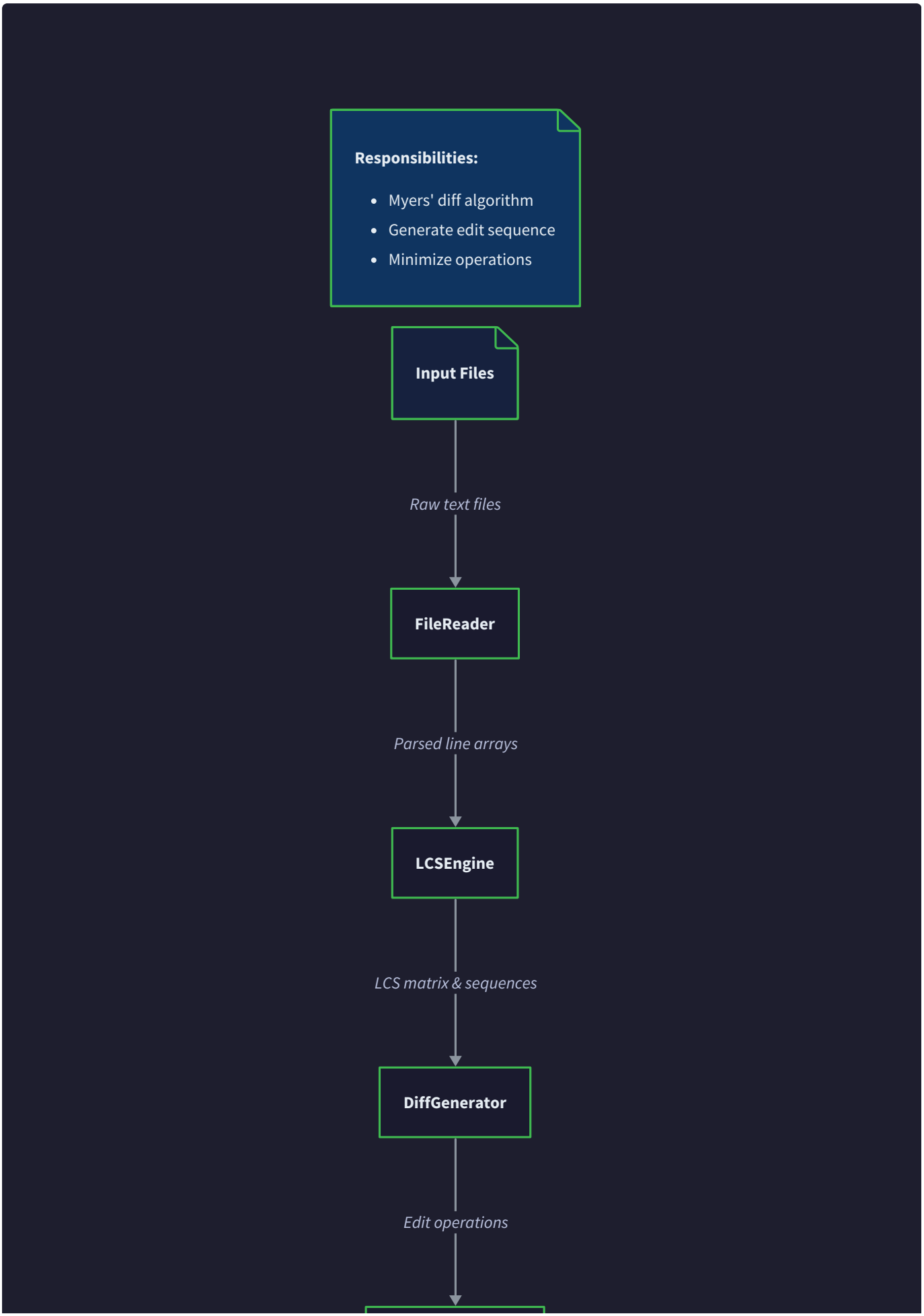
- Missing file headers (every diff needs --- and +++ lines)

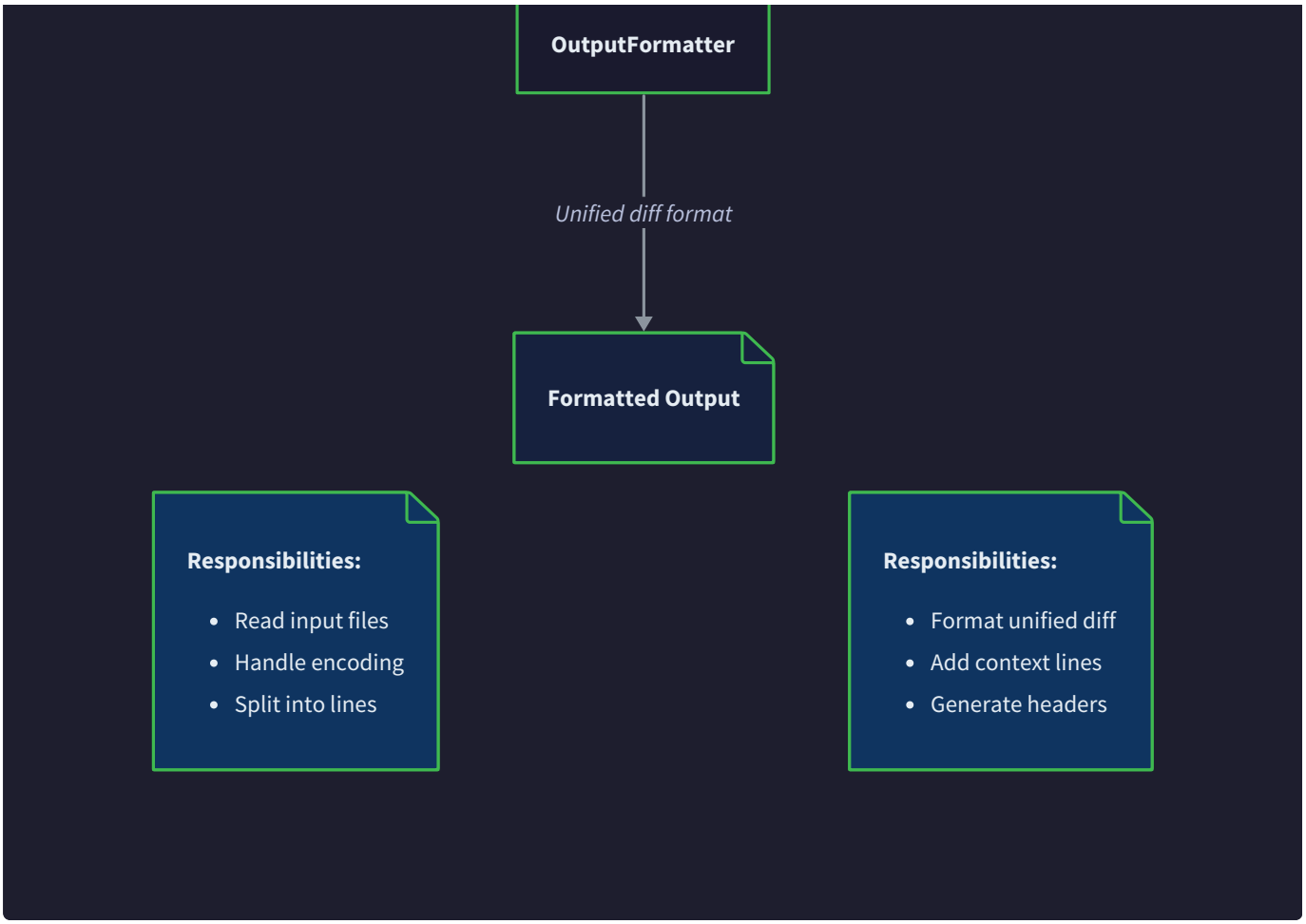
Interactions and Data Flow

> **Milestone(s):** All milestones — the component interactions and data flow span from line tokenization (Milestone 1) through LCS computation (Milestone 2), diff generation (Milestone 3), to final CLI output (Milestone 4)

Understanding how components communicate and data flows through our diff tool requires grasping the orchestrated sequence of operations that transforms two input files into meaningful, formatted output. Think of this process like an assembly line in a document processing facility, where each station (component) performs specialized operations on standardized work units, passing refined products to the next station until the final formatted comparison report emerges.

The interactions between `FileReader`, `LCSEngine`, `DiffGenerator`, and `OutputFormatter` follow a pipeline pattern where each component has clearly defined input and output contracts. This architectural choice ensures loose coupling between components while maintaining predictable data transformations throughout the diff computation process.


Responsibilities:
Myers' diff algorithm
Generate edit sequence
Minimize operations
Input Files
Raw text files
FileReader
Parsed line arrays
LCSEngine
LCS matrix & sequences
DiffGenerator
Edit operations



Component Communication

The communication between components follows a **message-passing interface pattern** where each component exposes well-defined methods that accept standardized data structures and return predictable outputs. This design choice eliminates tight coupling while ensuring type safety and clear responsibility boundaries.

> **Mental Model: Document Processing Pipeline**
>
> Imagine a document processing service where manuscripts arrive at different stations. The intake clerk standardizes format and breaks documents into pages. The analysis department compares pages to find similarities. The editorial team converts similarities into editing instructions. Finally, the publishing department formats instructions for readers. Each department has specific input requirements and output formats, communicating through standardized forms rather than direct collaboration.

Inter-Component Interface Contracts

The component interfaces define the message formats and method signatures that enable seamless communication throughout the diff pipeline:

Component	Input Interface	Output Interface	Communication Pattern
`FileReader`	File paths (strings)	`FileContent` structures	Synchronous method calls
`LCSEngine`	Two `Sequence` objects	`CommonSubsequence` structure	Synchronous computation
`DiffGenerator`	`CommonSubsequence` + original files	`List[Hunk]` structures	Synchronous transformation
`OutputFormatter`	`List[Hunk]` + formatting options	Formatted string output	Synchronous rendering

FileReader Communication Interface

The `FileReader` component exposes methods that abstract file system operations and return standardized `FileContent` structures. This interface insulates downstream components from file system complexities and encoding variations:

Method Name	Parameters	Returns	Description
`read_file_lines(filepath)`	`filepath: str`	`List[str]`	Reads file with encoding detection and normalization
`detect_file_encoding(filepath)`	`filepath: str`	`str`	Returns detected encoding (UTF-8 or Latin-1)
`create_file_content(filepath, raw_content, encoding, line_ending)`	`filepath: str, raw_content: str, encoding: str, line_ending: str`	`FileContent`	Factory method for validated content creation
`normalize_line_endings(content)`	`content: str`	`Tuple[str, str]`	Returns normalized content and detected ending type

The `FileReader` component communicates through `FileContent` structures that encapsulate all necessary metadata alongside the processed line arrays:

Field	Type	Description
`filepath`	`str`	Original file path for reference and error reporting
`lines`	`List[str]`	Normalized line array with consistent endings removed
`line_count`	`int`	Total number of lines including empty lines
`encoding`	`str`	Detected encoding (UTF-8 or LATIN-1)
`original_endings`	`str`	Original line ending type (LF, CRLF, or CR)

LCSEngine Communication Interface

The `LCSEngine` component receives `Sequence` objects (type alias for `List[str]`) and returns detailed `CommonSubsequence` structures that contain both the subsequence elements and their positions in the original sequences:

Method Name	Parameters	Returns	Description
`compute_lcs(sequence1, sequence2)`	`sequence1: Sequence, sequence2: Sequence`	`CommonSubsequence`	Main entry point with memory optimization
`build_lcs_matrix(seq1, seq2)`	`seq1: Sequence, seq2: Sequence`	`List[List[int]]`	Constructs dynamic programming matrix
`backtrack(matrix, seq1, seq2)`	`matrix: List[List[int]], seq1: Sequence, seq2: Sequence`	`List[str]`	Recovers actual LCS from completed matrix
`choose_strategy(seq1_len, seq2_len)`	`seq1_len: int, seq2_len: int`	`str`	Selects algorithm based on memory constraints

The `CommonSubsequence` structure provides comprehensive information about the longest common subsequence and its relationship to the original sequences:

Field	Type	Description
`elements`	`List[str]`	The actual longest common subsequence elements
`length`	`int`	Count of elements in the common subsequence
`positions1`	`List[int]`	Indices of LCS elements in first sequence
`positions2`	`List[int]`	Indices of LCS elements in second sequence

DiffGenerator Communication Interface

The `DiffGenerator` component transforms LCS results into structured edit operations and groups them into contextual hunks for readable output:

Method Name	Parameters	Returns	Description
`generate_diff(file1_lines, file2_lines, lcs, context_lines)`	`file1_lines: List[str], file2_lines: List[str], lcs: CommonSubsequence, context_lines: int`	`List[Hunk]`	Main entry point for diff generation
`lcs_to_edit_operations(lcs, file1_lines, file2_lines)`	`lcs: CommonSubsequence, file1_lines: List[str], file2_lines: List[str]`	`List[DiffLine]`	Converts LCS to flat edit operations
`diff_lines_to_hunks(lines, context_lines)`	`lines: List[DiffLine], context_lines: int`	`List[Hunk]`	Groups operations into contextual hunks
`create_diff_line(content, line_type, old_num, new_num)`	`content: str, line_type: LineType, old_num: Optional[int], new_num: Optional[int]`	`DiffLine`	Factory method for validated line creation

The `DiffLine` structure represents individual edit operations with complete metadata for formatting:

Field	Type	Description
`content`	`str`	The actual line content without diff prefixes
`line_type`	`LineType`	Operation type (UNCHANGED, ADDED, DELETED)
`old_line_num`	`Optional[int]`	Line number in original file (None for ADDED)
`new_line_num`	`Optional[int]`	Line number in new file (None for DELETED)

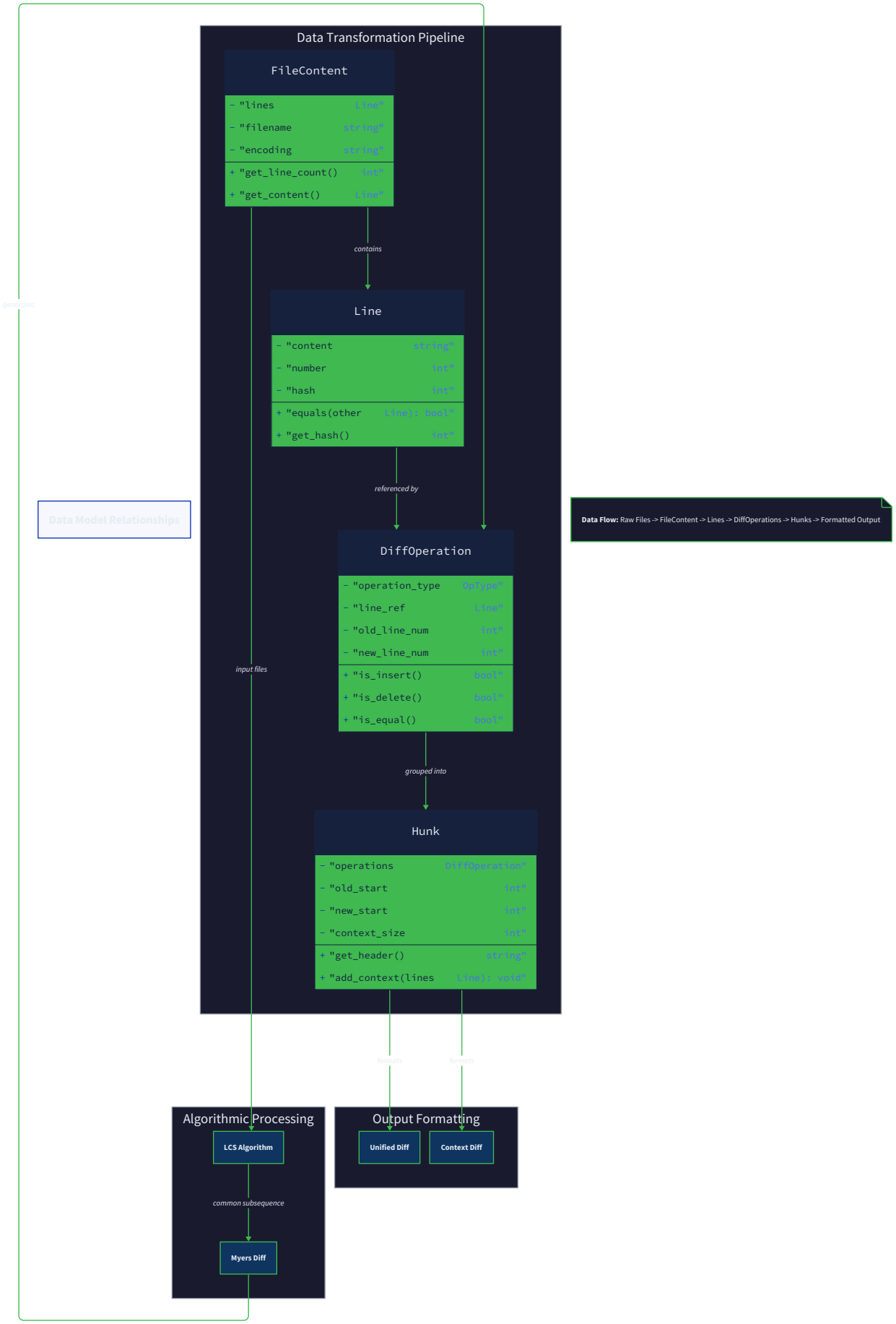
OutputFormatter Communication Interface

The `OutputFormatter` component receives structured hunks and produces formatted output with optional color support and various output modes:

Method Name	Parameters	Returns	Description
`format_unified_diff(file1, file2, hunks, output)`	`file1: FileContent, file2: FileContent, hunks: List[Hunk], output: TextIO`	`None`	Main entry point for generating unified diff output
`red(text)`	`text: str`	`str`	Format text with red ANSI codes if enabled
`green(text)`	`text: str`	`str`	Format text with green ANSI codes if enabled
`bold(text)`	`text: str`	`str`	Format text with bold ANSI codes if enabled
`is_color_enabled()`	None	`bool`	Check if color output is currently enabled

Decision: Synchronous Pipeline Architecture

- **Context**: Components need to communicate and pass data through multiple transformation stages
- **Options Considered**: Asynchronous message passing, synchronous method calls, shared memory with notifications
- **Decision**: Synchronous method calls with standardized data structures
- **Rationale**: Diff computation is inherently sequential (cannot generate hunks until LCS completes), synchronous calls simplify error handling and debugging, standardized structures ensure type safety
- **Consequences**: Simple to implement and reason about, clear error propagation, but no parallelization opportunities for large files


Data Transformation Pipeline
FileContent
- "lines Line"
- "filename string"
- "encoding string"
+ "get_line_count() int"
+ "get_content() Line"
contains
Line
- "content string"
- "number int"
- "hash int"
+ "equals(other Line): bool"
+ "get_hash() int"
referenced by
DiffOperation
- "operation_type OpType"
- "line_ref Line"
- "old_line_num int"
- "new_line_num int"
+ "is_insert() bool"
+ "is_delete() bool"
+ "is_equal() bool"
grouped into
Hunk
- "operations DiffOperation"
- "old_start int"
- "new_start int"
- "context_size int"
+ "get_header() string"
+ "add_context(lines Line): void"
generates
input files
formats
formats
Data Model Relationships
Data Flow: Raw Files -> FileContent -> Lines -> DiffOperations -> Hunks -> Formatted Output
Algorithmic Processing
LCS Algorithm
common subsequence
Myers Diff
Output Formatting
Unified Diff
Context Diff

Operation Sequence

The operation sequence follows a **linear pipeline pattern** where each component completes its processing before passing results to the next stage. This sequential approach matches the mathematical dependencies inherent in diff computation, where each stage requires complete results from the previous stage.

The overall flow proceeds through four distinct phases: **file preparation**, **similarity analysis**, **difference generation**, and **output formatting**. Each phase has specific responsibilities and produces intermediate results that serve as inputs for subsequent phases.

Phase 1: File Preparation and Normalization

The operation sequence begins when the main program invokes the `FileReader` component with two file paths. This phase focuses on converting raw file system data into normalized, comparable sequences:

1. **File System Access**: The `FileReader` component attempts to open both files, checking for existence and read permissions before proceeding with content extraction.
2. **Encoding Detection**: For each file, `detect_file_encoding()` is called to determine character encoding by attempting UTF-8 decoding first, falling back to Latin-1 for universal byte coverage.
3. **Content Reading**: The `read_file_lines()` method loads the entire file content into memory and applies encoding decoding based on the detected character set.
4. **Line Ending Normalization**: The `normalize_line_endings()` function identifies the original line ending type (LF, CRLF, or CR) and converts all endings to LF for consistent processing.
5. **Line Tokenization**: The `split_preserving_empty_lines()` method breaks content into individual line strings while maintaining empty lines that provide structural information.
6. **Structure Creation**: Two `FileContent` objects are created using `create_file_content()`, encapsulating the normalized line arrays along with metadata about encoding and original line endings.

The phase completes when both `FileContent` structures are available, containing clean line arrays ready for algorithmic comparison.

Decision Point: Memory vs. Streaming Trade-off

At this stage, the system has loaded both complete files into memory as line arrays. This approach enables random access during LCS computation but limits scalability for extremely large files:

Approach	Memory Usage	Algorithm Support	Implementation Complexity
Full Load	O(n) per file	Supports all LCS variants	Simple, enables backtracking
Streaming	O(1) per file	Limited algorithm options	Complex, requires specialized algorithms
Hybrid	O(k) sliding window	Most algorithms with modifications	Moderate, good for very large files

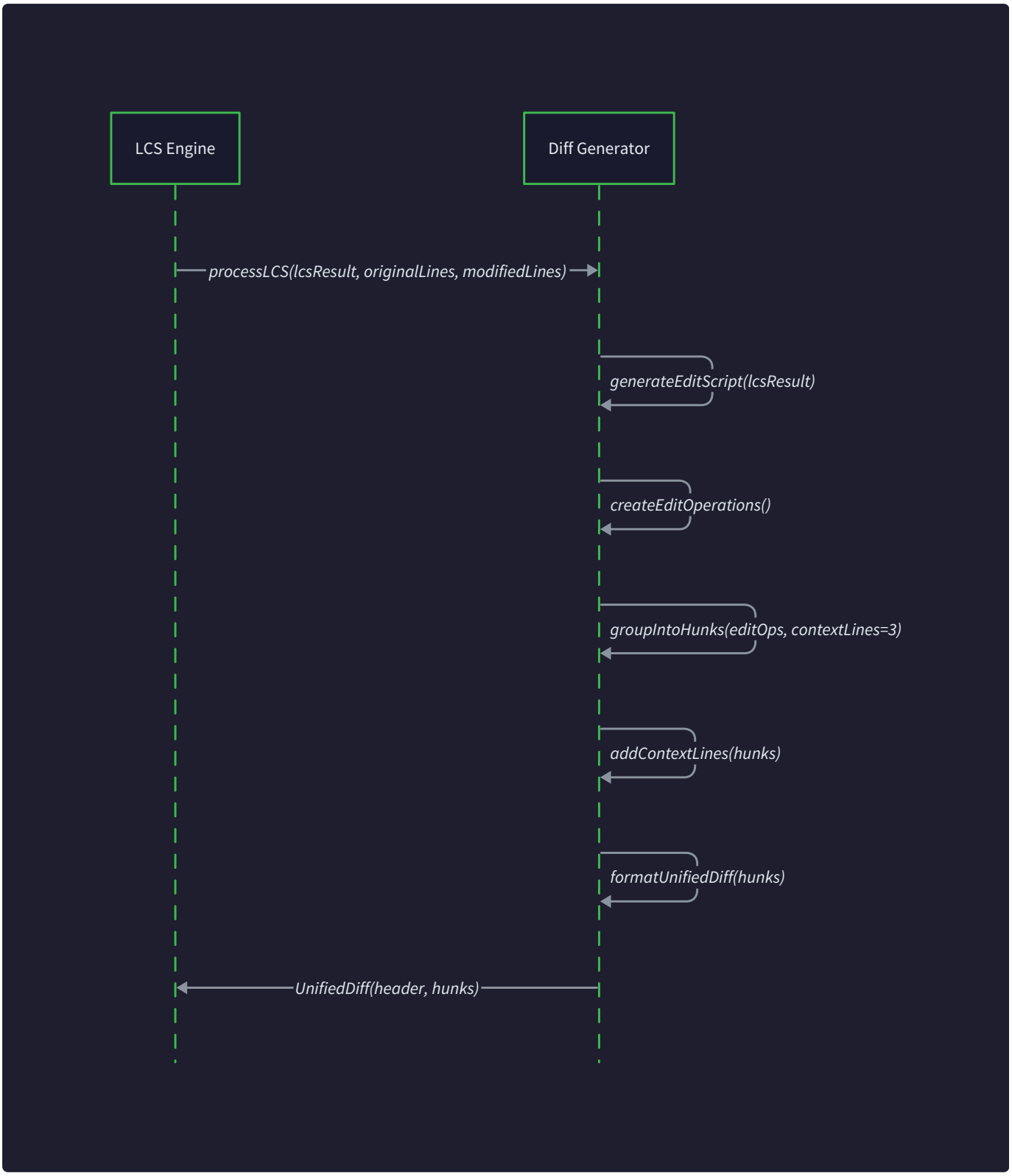
> The full-load approach was chosen because it enables efficient backtracking during LCS computation and supports future algorithm optimizations while maintaining reasonable memory usage for typical source files.

Phase 2: Similarity Analysis Through LCS

The similarity analysis phase transforms the normalized line sequences into a mathematical representation of their longest common subsequence, which forms the foundation for identifying unchanged regions:

1. **Strategy Selection**: The `LCSEngine` calls `choose_strategy()` to select an appropriate algorithm variant based on sequence lengths and available memory, choosing between standard dynamic programming, two-row optimization, or Hirschberg's space-efficient approach.
2. **Matrix Construction**: The `build_lcs_matrix()` method constructs the dynamic programming matrix by iterating through both sequences and applying the LCS recurrence relation to build up optimal subproblem solutions.
3. **Optimal Length Computation**: The matrix construction completes when the bottom-right cell contains the length of the longest common subsequence between the two input sequences.
4. **Subsequence Recovery**: The `backtrack()` method traverses the completed matrix from bottom-right to top-left, reconstructing the actual common subsequence elements by following optimal paths through the matrix.
5. **Position Tracking**: During backtracking, the algorithm records the original positions of each LCS element in both input sequences, enabling precise identification of unchanged regions.
6. **Result Packaging**: The `CommonSubsequence` structure is populated with the recovered elements, their positions in both sequences, and metadata about the computation process.

The similarity analysis phase completes when the `CommonSubsequence` structure contains complete information about shared content between the two files, ready for transformation into edit operations.



Phase 3: Difference Generation and Contextualization

The difference generation phase converts the mathematical LCS result into human-readable edit operations and groups them into contextual hunks for presentation:

1. **Edit Operation Generation**: The `lcs_to_edit_operations()` method processes the `CommonSubsequence` alongside the original line sequences to identify which lines should be marked as UNCHANGED, ADDED, or DELETED.

2. **Line-by-Line Classification**: The algorithm iterates through both input sequences simultaneously, using LCS position information to classify each line according to whether it appears in the common subsequence or represents an insertion or deletion.
3. **Metadata Assignment**: Each edit operation is wrapped in a `DiffLine` structure that includes the line content, operation type, and line numbers from both original files for accurate reporting.
4. **Hunk Formation**: The `diff_lines_to_hunks()` method analyzes the flat sequence of edit operations to identify natural groupings where changes occur close together, forming coherent hunks.
5. **Context Integration**: For each hunk, the algorithm adds configurable numbers of unchanged lines before and after the changes to provide context that helps readers understand the modifications.
6. **Hunk Optimization**: Adjacent hunks with small gaps between them are merged using `merge_adjacent_hunks()` to reduce fragmentation and improve readability of the final output.

The difference generation phase produces a list of `Hunk` structures, each containing a coherent group of changes with appropriate context for human consumption.

Decision Point: Context Line Strategy

The system must decide how many unchanged lines to include around each change group and when to merge nearby hunks:

Strategy	Context Lines	Hunk Merging	Readability	Output Size
Minimal	0-1 lines	Aggressive merging	Poor for large changes	Compact
Standard	3 lines	Merge if gap ≤ 6	Good balance	Moderate
Generous	5-10 lines	Conservative merging	Excellent readability	Large

> The standard strategy (3 context lines, merge if gap ≤ 6) was chosen to match conventional diff tool behavior and provide good readability without excessive output size.

Phase 4: Output Formatting and Presentation

The final phase transforms structured hunks into formatted output with appropriate visual styling and standard diff format compliance:

1. **Output Stream Preparation**: The `OutputFormatter` component determines the target output stream and checks for TTY capabilities to decide whether color formatting should be applied.
2. **File Header Generation**: The `_format_file_headers()` method creates the standard diff header lines that identify the compared files, including timestamps and file paths using the --- and +++ prefix convention.
3. **Hunk Processing**: For each hunk in the input list, `_format_hunk()` generates the complete hunk representation including the @@ header line and all constituent diff lines.

4. **Line Formatting**: Individual `DiffLine` objects are processed by `_format_diff_line()` to add appropriate prefixes (space for unchanged, - for deleted, + for added) and apply color formatting when enabled.
5. **Color Application**: When color output is enabled, the `red()`, `green()`, and `bold()` methods apply ANSI escape sequences to highlight deletions, additions, and headers respectively.
6. **Output Generation**: The formatted strings are written to the specified output stream, producing unified diff format that is compatible with standard tools and human-readable.

The operation sequence completes when all hunks have been formatted and written to the output stream, providing users with a comprehensive view of differences between the compared files.

Error Propagation Throughout the Pipeline

Error handling follows a **fail-fast strategy** where problems at any stage immediately terminate the pipeline and propagate detailed error information to the user:

Phase	Potential Failures	Detection Method	Recovery Action
File Preparation	Missing files, encoding errors, permission issues	Exception handling during file operations	Report specific file system error and exit
Similarity Analysis	Memory exhaustion, algorithm failures	Memory monitoring and computation validation	Switch to memory-efficient algorithm or report limitations
Difference Generation	Logic errors, inconsistent state	Assertion checks and data validation	Report internal error with diagnostic information
Output Formatting	I/O errors, formatting failures	Stream writing exception handling	Report output error and attempt graceful termination

> **Insight: Pipeline Simplicity vs. Performance**
>
> The sequential pipeline design prioritizes correctness and debuggability over performance optimization. While this approach prevents parallelization opportunities, it ensures that each component can be tested independently and errors can be precisely attributed to specific pipeline stages.

Performance Characteristics of the Pipeline

The overall performance characteristics are dominated by the LCS computation phase, which has O(mn) time complexity and determines the processing time for most file comparisons:

Phase	Time Complexity	Space Complexity	Dominant Factor
File Preparation	O(n + m)	O(n + m)	File I/O and line tokenization
Similarity Analysis	O(mn)	O(mn) or O(min(m,n))	Dynamic programming matrix
Difference Generation	O(n + m)	O(n + m)	Linear scan through sequences
Output Formatting	O(d) where d = diff lines	O(1)	I/O bandwidth to output stream

The pipeline design enables future optimizations such as streaming file reading, memory-efficient LCS variants, or parallel hunk formatting without requiring architectural changes to component interfaces.

Common Pitfalls

⚠️ Pitfall: Assuming Synchronous Operations Are Thread-Safe

Many developers assume that because the pipeline uses synchronous method calls, the components can be safely accessed from multiple threads. However, the `LCSEngine` maintains internal state in its matrix field, and the `OutputFormatter` may cache color settings.

Why this is wrong: Concurrent access to shared component instances can lead to matrix corruption during LCS computation or inconsistent color formatting. The symptoms include incorrect diff results or garbled output formatting.

How to fix: Create separate component instances for each diff operation, or add explicit synchronization around stateful operations. The recommended approach is to treat components as stateless service objects and pass all necessary state as method parameters.

⚠️ Pitfall: Ignoring Memory Implications of Full Pipeline

Developers often focus on optimizing individual components without considering the cumulative memory usage of the entire pipeline. Each phase creates intermediate data structures that exist simultaneously in memory.

Why this is wrong: For large files, the memory footprint includes original file content, normalized line arrays, LCS matrix, edit operations list, and formatted output buffers all existing at once. This can lead to memory exhaustion even when individual components seem efficient.

How to fix: Implement pipeline streaming where intermediate results are processed and discarded. Consider using generator patterns for large file processing, or implement explicit memory management with cleanup between phases.

⚠️ Pitfall: Inadequate Error Context Across Component Boundaries

When errors occur deep in the pipeline, developers often lose important context about which files were being processed or what stage failed. Generic exception handling obscures the specific operation that encountered problems.

Why this is wrong: Users receive unhelpful error messages like "encoding error" without knowing which file caused the problem or what encoding was detected. This makes debugging extremely difficult for end users.

How to fix: Wrap exceptions with additional context at each component boundary. Include file paths, detected encodings, sequence lengths, and operation phase in error messages. Create structured error types that carry diagnostic information through the pipeline.

Implementation Guidance

The interaction and data flow implementation requires careful orchestration of component method calls while maintaining clean separation of concerns and robust error handling throughout the pipeline.

A. Technology Recommendations

Component Interface	Simple Option	Advanced Option
Error Handling	Basic try/catch with print statements	Structured logging with error types and context
Data Validation	Manual assertions in component methods	Formal contracts with pre/post-condition checking
Memory Management	Default garbage collection	Explicit memory monitoring and optimization strategies
Pipeline Orchestration	Direct method calls in main function	Pipeline builder pattern with configurable stages

B. Recommended File Structure

```
diff-tool/
  main.py                          ← pipeline orchestration and CLI entry point
  components/
    file_reader.py                 ← FileReader component implementation
    lcs_engine.py                  ← LCSEngine component implementation
    diff_generator.py              ← DiffGenerator component implementation
    output_formatter.py            ← OutputFormatter component implementation
  data_model/
    types.py                       ← FileContent, CommonSubsequence, Hunk, DiffLine
    enums.py                       ← LineType, ColorMode constants
  utils/
    error_handling.py              ← Pipeline error types and context management
    memory_monitoring.py           ← Memory usage tracking and optimization
  tests/
    test_pipeline_integration.py   ← End-to-end pipeline testing
    test_component_interfaces.py   ← Component boundary and contract testing
```

C. Pipeline Orchestration Infrastructure

Complete pipeline coordinator that manages component interaction and error propagation:

```python
"""
Pipeline orchestrator that coordinates component interactions and manages
error propagation throughout the diff computation process.
"""

import sys
import traceback
from typing import List, Optional, TextIO
from dataclasses import dataclass
from components.file_reader import FileReader
from components.lcs_engine import LCSEngine
from components.diff_generator import DiffGenerator
from components.output_formatter import OutputFormatter
from data_model.types import FileContent, CommonSubsequence, Hunk

@dataclass
class PipelineContext:
    """Context information passed through pipeline stages for error reporting."""
    file1_path: str
    file2_path: str
    current_stage: str
    stage_progress: float
    memory_limit_mb: Optional[float] = None
    debug_mode: bool = False

class PipelineError(Exception):
    """Base exception for pipeline errors with context information."""
    def __init__(self, message: str, context: PipelineContext, cause: Exception = None):
        super().__init__(message)
```

```
        self.context = context
        self.cause = cause

class DiffPipeline:
    """Main pipeline coordinator managing component interactions."""

    def __init__(self, context_lines: int = 3, enable_color: bool = True):
        self.file_reader = FileReader()
        self.lcs_engine = LCSEngine()
        self.diff_generator = DiffGenerator()
        self.output_formatter = OutputFormatter(enable_color=enable_color)
        self.context_lines = context_lines

    def run_diff(self, file1_path: str, file2_path: str, output: TextIO = sys.stdout) -> int:
        """
        Execute complete diff pipeline and return exit code.
        Returns 0 if files identical, 1 if different, 2 if error.
        """
        context = PipelineContext(file1_path, file2_path, "initialization", 0.0)

        try:
            # Phase 1: File preparation
            context.current_stage = "file_preparation"
            context.stage_progress = 0.1
            file1_content, file2_content = self._prepare_files(context)

            # Phase 2: LCS computation
            context.current_stage = "lcs_computation"
```

```
            context.stage_progress = 0.3
            lcs_result = self._compute_similarity(file1_content, file2_content, context)

            # Phase 3: Diff generation
            context.current_stage = "diff_generation"
            context.stage_progress = 0.7
            hunks = self._generate_differences(file1_content, file2_content, lcs_result,
context)

            # Phase 4: Output formatting
            context.current_stage = "output_formatting"
            context.stage_progress = 0.9
            self._format_output(file1_content, file2_content, hunks, output, context)

            # Return appropriate exit code
            return 0 if len(hunks) == 0 else 1

        except PipelineError as e:
            self._handle_pipeline_error(e, output)
            return 2
        except Exception as e:
            pipeline_error = PipelineError(f"Unexpected error in {context.current_stage}",
context, e)
            self._handle_pipeline_error(pipeline_error, output)
            return 2

    def _prepare_files(self, context: PipelineContext) -> tuple[FileContent, FileContent]:
        """Phase 1: Read and normalize both input files."""
        try:
```

```
            file1_content = self.file_reader.read_file_content(context.file1_path)
            file2_content = self.file_reader.read_file_content(context.file2_path)
            return file1_content, file2_content
        except Exception as e:
            raise PipelineError(f"Failed to read input files: {str(e)}", context, e)

    def _compute_similarity(self, file1: FileContent, file2: FileContent, context:
PipelineContext) -> CommonSubsequence:
        """Phase 2: Compute longest common subsequence."""
        try:
            sequence1 = file1.lines
            sequence2 = file2.lines
            return self.lcs_engine.compute_lcs(sequence1, sequence2)
        except Exception as e:
            raise PipelineError(f"LCS computation failed: {str(e)}", context, e)

    def _generate_differences(self, file1: FileContent, file2: FileContent,
                              lcs: CommonSubsequence, context: PipelineContext) -> List[Hunk]:
        """Phase 3: Convert LCS to diff hunks."""
        try:
            return self.diff_generator.generate_diff(file1.lines, file2.lines, lcs,
self.context_lines)
        except Exception as e:
            raise PipelineError(f"Diff generation failed: {str(e)}", context, e)

    def _format_output(self, file1: FileContent, file2: FileContent,
                       hunks: List[Hunk], output: TextIO, context: PipelineContext):
        """Phase 4: Format and write output."""
        try:
            self.output_formatter.format_unified_diff(file1, file2, hunks, output)
```

```
        except Exception as e:
            raise PipelineError(f"Output formatting failed: {str(e)}", context, e)

    def _handle_pipeline_error(self, error: PipelineError, output: TextIO):
        """Handle pipeline errors with appropriate user messaging."""
        print(f"diff-tool: error in {error.context.current_stage}", file=sys.stderr)
        print(f"Files: {error.context.file1_path}, {error.context.file2_path}",
file=sys.stderr)
        print(f"Error: {str(error)}", file=sys.stderr)

        if error.context.debug_mode and error.cause:
            print("Debug traceback:", file=sys.stderr)
            traceback.print_exception(type(error.cause), error.cause,
error.cause.__traceback__)
```

D. Component Interface Contracts

Interface validation utilities that ensure components conform to expected contracts:

```python
"""
Component interface validation and contract enforcement for pipeline integrity.
"""

from abc import ABC, abstractmethod
from typing import List, Protocol
from data_model.types import FileContent, CommonSubsequence, Hunk, Sequence

class FileReaderInterface(Protocol):
    """Contract for file reading components."""

    def read_file_content(self, filepath: str) -> FileContent:
        """Read file with encoding detection and normalization."""
        pass

    def detect_file_encoding(self, filepath: str) -> str:
        """Detect file encoding, returns 'UTF-8' or 'LATIN-1'."""
        pass

class LCSEngineInterface(Protocol):
    """Contract for LCS computation components."""

    def compute_lcs(self, sequence1: Sequence, sequence2: Sequence) -> CommonSubsequence:
        """Compute longest common subsequence with position tracking."""
        pass

    def choose_strategy(self, seq1_len: int, seq2_len: int) -> str:
        """Select algorithm strategy based on input size."""
        pass
```

```python
class DiffGeneratorInterface(Protocol):
    """Contract for diff generation components."""

    def generate_diff(self, file1_lines: List[str], file2_lines: List[str],
                      lcs: CommonSubsequence, context_lines: int) -> List[Hunk]:
        """Generate contextual diff hunks from LCS result."""
        pass

class OutputFormatterInterface(Protocol):
    """Contract for output formatting components."""

    def format_unified_diff(self, file1: FileContent, file2: FileContent,
                            hunks: List[Hunk], output) -> None:
        """Format hunks as unified diff with optional color."""
        pass

def validate_component_contracts():
    """Runtime validation that components implement required interfaces."""
    # This would contain runtime checks for interface compliance
    pass
```

E. Core Pipeline Logic Skeleton

Main pipeline orchestration with detailed TODO comments for implementation:

```python
def run_diff_pipeline(file1_path: str, file2_path: str, context_lines: int = 3) -> int:
    """
    Execute the complete diff pipeline from file input to formatted output.
    Returns exit code: 0 if identical, 1 if different, 2 if error.
    """

    # TODO 1: Initialize all four pipeline components
    # - Create FileReader instance for file operations
    # - Create LCSEngine instance for similarity computation
    # - Create DiffGenerator instance for edit operation generation
    # - Create OutputFormatter instance with color settings

    # TODO 2: Phase 1 - File Preparation
    # - Call file_reader.read_file_content() for both files
    # - Handle file system errors (missing files, permissions, encoding)
    # - Store FileContent structures for both input files
    # - Validate that both files were read successfully

    # TODO 3: Phase 2 - LCS Computation
    # - Extract line sequences from FileContent.lines
    # - Call lcs_engine.compute_lcs() with both sequences
    # - Handle potential memory exhaustion for large files
    # - Store CommonSubsequence result for next phase

    # TODO 4: Phase 3 - Diff Generation
    # - Call diff_generator.generate_diff() with files, LCS, and context_lines
    # - Handle edge cases (no common lines, identical files)
    # - Store List[Hunk] result for formatting
```

```
    # TODO 5: Phase 4 - Output Formatting
    # - Call output_formatter.format_unified_diff() with all hunks
    # - Handle output stream errors and color formatting
    # - Ensure proper unified diff format compliance

    # TODO 6: Exit Code Logic
    # - Return 0 if hunks list is empty (files identical)
    # - Return 1 if hunks exist (files different)
    # - Return 2 if any phase encountered errors
    # Hint: len(hunks) == 0 indicates identical files

    pass  # Replace with implementation
```

F. Error Context Management

PYTHON

```python
"""Error handling utilities for maintaining context through pipeline stages."""


class ErrorContext:
    """Tracks pipeline execution context for detailed error reporting."""


    def __init__(self, file1_path: str, file2_path: str):
        self.file1_path = file1_path
        self.file2_path = file2_path
        self.current_phase = "initialization"
        self.phase_data = {}


    def enter_phase(self, phase_name: str, **phase_info):
        """Enter new pipeline phase with context information."""
        # TODO: Store phase name and optional phase-specific data
        # TODO: Record timestamp for performance monitoring
        pass


    def add_context(self, key: str, value):
        """Add contextual information for current phase."""
        # TODO: Store key-value context data for error reporting
        pass


    def format_error_message(self, error: Exception) -> str:
        """Format error with full pipeline context."""
        # TODO: Create comprehensive error message including:
        # - File paths being processed
        # - Current pipeline phase
        # - Phase-specific context data
```

```
        # - Original exception message
        pass
```

G. Milestone Checkpoints

After implementing the component interactions and data flow:

Checkpoint 1: Pipeline Assembly

- Run: `python main.py file1.txt file2.txt`
- Expected: Program executes without import errors and attempts file reading
- Verify: Error messages mention specific pipeline phases when components are missing
- Signs of problems: Import errors, component instantiation failures, missing method errors

Checkpoint 2: Component Communication

- Create two simple text files with known differences
- Run pipeline and verify each component receives expected inputs
- Expected: FileReader produces FileContent, LCSEngine receives line sequences
- Signs of problems: Type errors at component boundaries, None values passed between stages

Checkpoint 3: Error Propagation

- Test with non-existent file, binary file, and permission-denied file
- Expected: Specific error messages identifying problem file and pipeline stage
- Verify: Exit codes match conventions (0=same, 1=different, 2=error)
- Signs of problems: Generic error messages, wrong exit codes, unhandled exceptions

H. Debugging Pipeline Issues

Symptom	Likely Cause	How to Diagnose	Fix
Pipeline hangs indefinitely	LCS matrix too large for memory	Monitor memory usage during execution	Implement memory-efficient LCS strategy
Wrong exit codes returned	Logic error in hunk counting	Print len(hunks) before return statement	Fix identical file detection logic
Components can't communicate	Interface mismatch between components	Check method signatures and return types	Ensure consistent data structure usage
Memory usage grows without bound	Intermediate results not freed	Profile memory usage by phase	Add explicit cleanup between phases
Error messages lack context	Generic exception handling	Add logging at each component boundary	Implement structured error context

Error Handling and Edge Cases

> **Milestone(s):** All milestones — robust error handling is essential throughout line tokenization (Milestone 1), LCS computation (Milestone 2), diff generation (Milestone 3), and CLI output (Milestone 4)

Think of error handling in a diff tool like building a bridge across a river. The bridge must withstand not just normal traffic, but also storms, floods, and unexpected loads. Similarly, our diff tool must handle not just well-formed text files, but also missing files, permission denied errors, binary content, memory exhaustion, and malformed inputs. A diff tool that crashes on edge cases is like a bridge that collapses in bad weather — it fails exactly when users need it most.

The complexity of error handling in text comparison tools comes from the intersection of multiple failure domains: the file system layer can fail with I/O errors, the encoding detection layer can encounter binary or malformed data, the algorithm layer can exhaust memory on large inputs, and the output layer can fail when writing to pipes or terminals. Each layer must fail gracefully while preserving enough context for meaningful error messages and recovery strategies.

Our error handling strategy follows a **fail-fast principle** with **contextual error propagation**. Rather than attempting to recover from fundamental errors like missing files, we detect problems early and provide clear diagnostic information. This approach prevents cascading failures where one component's silent error leads to mysterious failures in downstream components.

> **Design Principle:** Error messages should answer three questions: What went wrong? Why did it happen? What can the user do about it?

File System Errors

The `FileReader` component operates at the boundary between our application and the unreliable external world of file systems. File system operations can fail in numerous ways, each requiring different detection strategies and recovery approaches.

Encoding Detection Failures

Encoding detection represents one of the most subtle error scenarios in text processing. The `detect_file_encoding` function attempts UTF-8 decoding first, falling back to Latin-1 if UTF-8 fails. However, this process can encounter several failure modes that require careful handling.

Failure Mode	Detection Strategy	Recovery Approach	Error Context
Binary File	UTF-8 decode raises `UnicodeDecodeError` with null bytes	Detect binary content and refuse to process	"File appears to be binary (contains null bytes at position X)"
Truncated UTF-8	Incomplete multibyte sequence at file end	Check for incomplete sequences in last 4 bytes	"File contains truncated UTF-8 sequence at end"
Mixed Encodings	UTF-8 decode succeeds partially then fails	Track decode position and report first failure	"File contains mixed encodings starting at line X"
Zero-Length File	File exists but contains no data	Check file size before encoding detection	"Empty file - no content to compare"
Very Large File	File size exceeds available memory	Check file size against memory limits	"File too large (X MB) - exceeds memory limit (Y MB)"

The encoding detection process must distinguish between recoverable encoding issues and fundamental problems. Binary files containing executable code or images should be rejected with clear error messages, while text files with minor encoding issues might be processable with warnings.

File Access Permission Errors

File permission errors occur at multiple points in the pipeline and require different handling strategies depending on when they're encountered:

Access Phase	Error Scenarios	Detection Method	Recovery Strategy
Initial Open	File doesn't exist, permission denied, directory instead of file	`os.open()` raises `FileNotFoundError`, `PermissionError`, `IsADirectoryError`	Immediate failure with specific error message
Size Check	File exists but `stat()` permission denied	`os.stat()` raises `PermissionError`	Skip size optimization, attempt direct read
Content Read	Partial read permission (some bytes readable)	`file.read()` returns fewer bytes than expected	Report partial read with byte counts
Lock Conflicts	Another process has exclusive lock on file	Read operation blocks or raises OS-specific error	Timeout with suggestion to check file usage

Permission errors must be reported with sufficient context for users to understand the problem. Rather than generic "access denied" messages, we provide specific information about which operation failed and potential

causes.

Line Ending and Structure Errors

The line tokenization process can encounter malformed file structures that challenge our assumptions about text format:

Structure Issue	Detection Approach	Handling Strategy	User Impact
Mixed Line Endings	Track `\n`, `\r\n`, `\r` patterns across file	Normalize to LF, warn about mixed format	"Warning: File contains mixed line endings (X Unix, Y Windows, Z Mac)"
Very Long Lines	Line exceeds reasonable length threshold (e.g., 1MB)	Split at threshold with continuation marker	"Warning: Lines truncated at 1MB limit (affects lines X, Y, Z)"
Null Characters	Embedded null bytes in otherwise text file	Count nulls and report positions	"Warning: Text file contains X null characters (binary data?)"
Control Characters	Non-printable characters except common whitespace	Track unusual control character usage	"Warning: File contains unusual control characters"

The `split_preserving_empty_lines` function must handle these edge cases while maintaining the structural integrity needed for accurate line-by-line comparison. Each anomaly should be logged with sufficient detail for debugging while not preventing comparison when possible.

> **Decision: Graceful Degradation for File Structure Issues**
>
> - **Context**: Text files in practice contain various anomalies that shouldn't prevent comparison
> - **Options Considered**: Strict rejection, silent normalization, graceful degradation with warnings
> - **Decision**: Graceful degradation with detailed warnings
> - **Rationale**: Users often need to compare imperfect files; warnings preserve utility while flagging issues
> - **Consequences**: Requires careful anomaly detection and clear warning messages, but maximizes tool usefulness

Algorithm Edge Cases

The LCS algorithm and diff generation process encounter mathematical and computational edge cases that can cause subtle bugs or catastrophic failures. These scenarios often arise from boundary conditions in the dynamic programming algorithm or resource exhaustion during computation.

Empty and Identical File Scenarios

Empty files and identical files represent the mathematical boundaries of the diff problem space. These cases must be handled efficiently without invoking the full LCS computation:

Scenario	Detection Point	Optimization Strategy	Expected Output
Both Files Empty	After line tokenization, both `line_count` == 0	Skip LCS computation entirely	"Files are identical (both empty)"
One File Empty	One `line_count` == 0, other > 0	Generate pure addition or deletion diff	All lines marked as ADDED or DELETED
Files Identical	Early hash comparison or line-by-line check	Skip LCS computation	"Files are identical"
Single Line Files	Both `line_count` == 1	Direct comparison without matrix	Either identical or single-line diff

The `LCSEngine` component must detect these scenarios before constructing the dynamic programming matrix. Attempting to build a matrix for empty sequences can cause index errors, while building matrices for identical files wastes computational resources.

Memory Exhaustion Scenarios

Large file comparisons can exhaust available memory during LCS matrix construction. The standard dynamic programming approach requires O(m×n) space, which becomes prohibitive for files with hundreds of thousands of lines:

File Size Scenario	Memory Requirements	Detection Strategy	Mitigation Approach
Small Files (<1000 lines each)	<1MB matrix memory	No detection needed	Standard O(mn) algorithm
Medium Files (1000-10000 lines)	1-100MB matrix memory	Check available system memory	Standard algorithm with progress monitoring
Large Files (10000-100000 lines)	100MB-10GB matrix memory	Pre-calculate memory needs	Switch to two-row optimization
Huge Files (>100000 lines)	>10GB matrix memory	Memory requirement exceeds limits	Switch to Hirschberg's algorithm or refuse

The `choose_strategy` method in `LCSEngine` must evaluate these scenarios before beginning computation. Memory estimation must account for the matrix storage plus additional structures like the backtracking path and intermediate results.

LCS Computation Edge Cases

The dynamic programming algorithm itself can encounter numerical and logical edge cases that lead to incorrect results or infinite loops:

Edge Case	Manifestation	Detection Method	Correction Strategy
Off-by-One Indexing	Matrix bounds errors, incorrect LCS length	Boundary condition testing with known inputs	Careful index validation in matrix construction
No Common Lines	LCS length is 0, backtracking finds empty sequence	Check `lcs.length == 0` after computation	Generate pure addition/deletion diff
All Lines Identical	LCS equals entire shorter file	`lcs.length == min(len(seq1), len(seq2))`	Optimize to avoid full matrix computation
Repeated Line Patterns	Multiple valid LCS paths of same length	Backtracking algorithm must choose consistently	Use consistent tie-breaking rules

The backtracking algorithm in particular must handle cases where multiple paths through the matrix yield equally valid longest common subsequences. Consistency in tie-breaking ensures that repeated runs on the same input produce identical output.

Diff Generation Edge Cases

Converting LCS results to edit operations and hunks introduces additional edge cases related to line numbering and hunk boundary detection:

Generation Issue	Problem Manifestation	Detection Approach	Resolution Method
Line Number Misalignment	Hunk headers show wrong line ranges	Validate line counts against actual content	Recalculate line numbers during hunk formation
Empty Hunks	Hunk contains no actual changes	Check that `hunk.lines` contains at least one ADD or DELETE	Filter empty hunks before output
Overlapping Context	Adjacent hunks have overlapping context lines	Calculate context overlap during hunk merging	Merge hunks or adjust context boundaries
File Boundary Context	Context lines requested beyond file start/end	Check context line numbers against file bounds	Clamp context to file boundaries

The `diff_lines_to_hunks` function must validate hunk consistency after formation. Each hunk's line count claims must match the actual number of lines in its content, and line numbering must remain consistent across hunk boundaries.

Resource Limit Enforcement

Beyond memory exhaustion, the diff tool must handle other resource constraints that can cause degraded performance or failures:

Resource Constraint	Limit Detection	Graceful Degradation	User Communication
CPU Time	Track computation elapsed time	Interrupt after timeout, provide partial results	"Computation timed out after X seconds - files too complex"
File Handle Limits	Monitor open file descriptors	Close files promptly, reopen as needed	"System file handle limit reached"
Disk Space	Check available space before writing output	Stream output, avoid temporary files	"Insufficient disk space for output"
Network File Systems	Detect network mount points	Warn about potential latency	"Warning: Comparing files over network - may be slow"

The `PerformanceMonitor` component tracks these resource constraints and provides early warnings when limits are approached. Rather than silent failures or mysterious hangs, users receive clear information about resource constraints and potential workarounds.

> **⚠️ Pitfall: Silent Memory Exhaustion**
>
> A common mistake is allowing the LCS matrix construction to consume all available memory, causing the system to swap heavily or the process to be killed by the OS. This manifests as extreme slowness or sudden termination without error messages. The fix is to estimate memory requirements before allocation and either use memory-efficient algorithms or refuse to process files that exceed reasonable limits. Always check `matrix_size * sizeof(int) < available_memory` before proceeding.

> **⚠️ Pitfall: Integer Overflow in Line Counting**
>
> Very large files can cause integer overflow when calculating line numbers or matrix indices, leading to negative line numbers or array bounds errors. This typically occurs with files containing millions of lines. The fix is to use appropriate integer types (64-bit integers in most cases) and validate that line counts remain within reasonable bounds. Check for `line_count > MAX_SAFE_LINES` and refuse to process impossibly large files.

Implementation Guidance

The error handling implementation must balance comprehensive coverage with maintainable code structure. Python's exception hierarchy and context managers provide excellent tools for building robust error handling.

Technology Recommendations

Error Handling Aspect	Simple Option	Advanced Option
Exception Types	Built-in exceptions (`FileNotFoundError`, `MemoryError`)	Custom exception hierarchy with error codes
Memory Monitoring	`psutil.virtual_memory()`	Custom memory tracking with allocation hooks
File Operations	Standard `open()` with try/except	`pathlib.Path` with comprehensive error mapping
Logging	Python `logging` module	Structured logging with JSON output
Resource Limits	Manual checks before operations	`resource` module with automatic enforcement

File Structure for Error Handling

```
diff_tool/
  src/
    exceptions.py           ← Custom exception definitions
    error_handlers.py       ← Error detection and recovery strategies
    resource_monitor.py     ← Memory and resource tracking
    file_reader.py          ← File operations with error handling
    lcs_engine.py           ← Algorithm with memory management
    diff_generator.py       ← Generation with validation
    output_formatter.py     ← Output with I/O error handling
  tests/
    test_error_cases.py     ← Comprehensive error scenario testing
    fixtures/
      binary_file.exe       ← Test fixture for binary detection
      empty_file.txt        ← Empty file test case
      huge_file.txt         ← Large file for memory testing
```

Custom Exception Hierarchy (Complete Implementation)

```python
"""
Custom exceptions for diff tool with structured error information.
Provides context and recovery suggestions for different failure modes.
"""

class DiffToolError(Exception):
    """Base exception for all diff tool errors with context tracking."""

    def __init__(self, message, context=None, suggestion=None):
        super().__init__(message)
        self.context = context or {}
        self.suggestion = suggestion
        self.error_code = getattr(self.__class__, 'ERROR_CODE', 'UNKNOWN')

    def format_error(self):
        """Format error with context and suggestions for user display."""
        lines = [f"Error: {self}"]

        if self.context:
            lines.append("Context:")
            for key, value in self.context.items():
                lines.append(f"  {key}: {value}")

        if self.suggestion:
            lines.append(f"Suggestion: {self.suggestion}")

        return "\n".join(lines)

class FileSystemError(DiffToolError):
```

```
    """File system operation failures with specific error codes."""
    ERROR_CODE = 'FILE_SYSTEM'

class EncodingError(DiffToolError):
    """File encoding detection and processing failures."""
    ERROR_CODE = 'ENCODING'

class AlgorithmError(DiffToolError):
    """LCS computation and diff generation failures."""
    ERROR_CODE = 'ALGORITHM'

class ResourceError(DiffToolError):
    """Memory, time, or other resource exhaustion."""
    ERROR_CODE = 'RESOURCE'

class OutputError(DiffToolError):
    """Output formatting and writing failures."""
    ERROR_CODE = 'OUTPUT'
```

Resource Monitor (Complete Implementation)

PYTHON

```python
"""
Resource monitoring and limit enforcement for diff operations.
Prevents memory exhaustion and provides early warnings for resource constraints.
"""

import psutil
import time
from typing import Optional, Dict, Any

class ResourceMonitor:
    """Monitors memory usage, computation time, and other system resources."""

    def __init__(self, memory_limit_mb: float = 1024, time_limit_seconds: int = 300):
        self.memory_limit_mb = memory_limit_mb
        self.time_limit_seconds = time_limit_seconds
        self.start_time: Optional[float] = None
        self.peak_memory_mb = 0.0

    def start_monitoring(self):
        """Begin resource monitoring for an operation."""
        self.start_time = time.time()
        self.peak_memory_mb = self._get_memory_usage_mb()

    def check_limits(self) -> Optional[ResourceError]:
        """Check if resource limits have been exceeded."""
        if self.start_time is None:
            return None

        # Check time limit
```

```
        elapsed = time.time() - self.start_time
        if elapsed > self.time_limit_seconds:
            return ResourceError(
                f"Operation timed out after {elapsed:.1f} seconds",
                context={"time_limit": self.time_limit_seconds, "elapsed": elapsed},
                suggestion="Try smaller files or increase time limit"
            )

        # Check memory limit
        current_memory = self._get_memory_usage_mb()
        self.peak_memory_mb = max(self.peak_memory_mb, current_memory)

        if current_memory > self.memory_limit_mb:
            return ResourceError(
                f"Memory usage ({current_memory:.1f} MB) exceeds limit
({self.memory_limit_mb:.1f} MB)",
                context={"memory_used": current_memory, "memory_limit": self.memory_limit_mb},
                suggestion="Try using memory-optimized algorithms or increase memory limit"
            )

        return None

    def estimate_matrix_memory(self, rows: int, cols: int) -> float:
        """Estimate memory requirements for LCS matrix in MB."""
        # Each matrix cell is typically 4 bytes (32-bit int)
        bytes_needed = rows * cols * 4
        return bytes_needed / (1024 * 1024)
```

```
    def _get_memory_usage_mb(self) -> float:
        """Get current process memory usage in MB."""
        process = psutil.Process()
        return process.memory_info().rss / (1024 * 1024)


    def get_stats(self) -> Dict[str, Any]:
        """Get monitoring statistics for reporting."""
        return {
            "peak_memory_mb": self.peak_memory_mb,
            "elapsed_seconds": time.time() - self.start_time if self.start_time else 0,
            "memory_limit_mb": self.memory_limit_mb,
            "time_limit_seconds": self.time_limit_seconds
        }
```

Core Logic Skeletons with Error Handling

```python
def detect_file_encoding(filepath: str) -> str:
    """
    Detect file encoding trying UTF-8 then Latin-1 with comprehensive error handling.
    Returns encoding string or raises EncodingError with specific failure details.
    """
    # TODO 1: Check if file exists and is readable - raise FileSystemError with context if not
    # TODO 2: Check file size - if zero bytes, raise EncodingError indicating empty file
    # TODO 3: Read first 1024 bytes for encoding detection without loading entire file
    # TODO 4: Attempt UTF-8 decoding - catch UnicodeDecodeError and check for null bytes (binary file)
    # TODO 5: If UTF-8 fails, attempt Latin-1 decoding - Latin-1 should always succeed for binary data
    # TODO 6: If both fail (shouldn't happen), raise EncodingError with detection failure details
    # TODO 7: For successful detection, validate by trying to decode a larger sample
    # Hint: Use try/except blocks for each encoding attempt and preserve original exception details
    pass

def compute_lcs_with_limits(sequence1: Sequence, sequence2: Sequence, monitor: ResourceMonitor) -> CommonSubsequence:
    """
    Compute LCS with resource monitoring and graceful degradation on limits.
    Switches algorithms based on input size and available resources.
    """
    # TODO 1: Start resource monitoring and get initial memory baseline
    # TODO 2: Handle empty sequence edge cases - return appropriate CommonSubsequence immediately
    # TODO 3: Estimate memory requirements for full matrix algorithm
    # TODO 4: If memory estimate exceeds limits, choose memory-efficient algorithm
    # TODO 5: Periodically check resource limits during matrix construction
```

```
    # TODO 6: If limits exceeded during computation, clean up and raise ResourceError with
context

    # TODO 7: Validate result consistency - LCS length should not exceed shorter sequence

    # TODO 8: Return CommonSubsequence with computation statistics attached

    # Hint: Use monitor.check_limits() after each major step, clean up matrix on failure

    pass


def generate_diff_with_validation(file1_lines: List[str], file2_lines: List[str],

                                  lcs: CommonSubsequence, context_lines: int) -> List[Hunk]:

    """

    Generate diff hunks with comprehensive validation and error recovery.

    Validates line numbering consistency and hunk boundary correctness.

    """

    # TODO 1: Validate LCS result consistency against input files

    # TODO 2: Convert LCS to edit operations, tracking line numbers carefully

    # TODO 3: Validate that edit operations account for all lines in both files

    # TODO 4: Group operations into hunks with requested context

    # TODO 5: Validate each hunk's line count claims against actual content

    # TODO 6: Check for overlapping context between adjacent hunks

    # TODO 7: Merge or adjust hunks to resolve context overlaps

    # TODO 8: Final validation pass - ensure line numbers are monotonic and complete

    # Hint: Keep running totals of processed lines to catch numbering errors early

    pass
```

Milestone Checkpoints for Error Handling

After implementing error handling for each milestone, verify these specific behaviors:

Milestone 1 Checkpoint - File Reading Errors:

- Command: `python diff_tool.py nonexistent.txt other.txt`
- Expected: Clear error message about missing file, exit code 1
- Command: `python diff_tool.py /dev/null /dev/null` (Unix) or empty files

- Expected: "Both files are empty" message, exit code 0
- Command: `python diff_tool.py binary_file.exe text_file.txt`
- Expected: Error about binary file detection with null byte position

Milestone 2 Checkpoint - Algorithm Limits:

- Create test files with 10,000+ lines each
- Command: `python diff_tool.py huge1.txt huge2.txt`
- Expected: Either successful completion with memory monitoring or graceful failure with resource limit message
- Verify memory usage doesn't exceed system limits
- Check that empty file comparisons complete instantly without matrix construction

Milestone 3 Checkpoint - Diff Validation:

- Create files with edge cases: no common lines, all identical lines, single character differences
- Verify hunk line counts match content
- Check that all input lines appear exactly once in diff output
- Validate line numbers are consistent and monotonic

Milestone 4 Checkpoint - CLI Error Handling:

- Test all error scenarios produce appropriate exit codes (0 for success, 1 for differences, 2 for errors)
- Verify color codes don't appear when output is piped: `python diff_tool.py file1.txt file2.txt | cat`
- Check that Ctrl+C interruption cleans up resources properly

Debugging Tips for Error Handling

Symptom	Likely Cause	Diagnosis Steps	Fix Strategy
Tool crashes silently	Unhandled exception or resource exhaustion	Check system logs, monitor memory usage during run	Add try/catch blocks around main operations, implement resource monitoring
"Binary file" error on text files	Encoding detection failing, or file contains null bytes	Examine file with hex editor, check for encoding issues	Improve encoding detection logic, handle mixed encodings gracefully
Line numbers in diff output are wrong	Off-by-one errors in hunk generation	Compare line numbers in hunks against original files manually	Validate line counting logic, use 1-indexed line numbers consistently
Memory usage grows without bound	LCS matrix not being garbage collected	Profile memory usage during algorithm execution	Implement matrix cleanup, use memory-efficient algorithms for large inputs
Tool hangs on large files	Infinite loop in algorithm or excessive memory swapping	Use profiler to identify bottleneck, monitor system resources	Add progress monitoring, implement timeouts, detect swap thrashing

Testing Strategy

> **Milestone(s):** All milestones — comprehensive testing ensures correctness throughout line tokenization (Milestone 1), LCS computation (Milestone 2), diff generation (Milestone 3), and CLI output (Milestone 4)

Testing a diff tool requires a systematic approach that validates both algorithmic correctness and practical usability. The core challenge lies in verifying that complex dynamic programming algorithms produce correct results while handling the myriad edge cases that arise from real-world file formats and user scenarios.

Mental Model: Quality Assurance Chain

Think of testing a diff tool like quality assurance in a publishing house that compares manuscript versions. The QA process has multiple checkpoints: first, verify that each editor (component) correctly identifies their assigned changes; then, confirm that all editorial notes (diff operations) accurately reflect the differences; finally, ensure the published comparison (formatted output) is readable and follows industry standards. Each checkpoint catches different types of errors, and the chain is only as strong as its weakest link.

The testing strategy mirrors this approach by validating each component individually, then verifying their integration produces correct end-to-end behavior. Just as a publishing house maintains reference examples of well-executed comparisons, our test suite relies on golden files and known-correct diff outputs to validate algorithmic behavior.

Unit Testing Approach

Unit testing for a diff tool requires careful construction of test cases that isolate component behavior while exercising the full range of inputs each component might encounter. The strategy focuses on testing individual components with known input-output pairs, systematic edge case coverage, and algorithmic validation using reference implementations.

Component Isolation Strategy

Each component in the diff pipeline requires different testing approaches based on its algorithmic complexity and interaction patterns. The `FileReader` component primarily handles I/O and text processing, making it suitable for file-based testing with various encodings and formats. The `LCSEngine` involves complex dynamic programming algorithms that benefit from mathematical verification against known solutions. The `DiffGenerator` transforms algorithmic results into structured output, requiring format validation and context handling verification. The `OutputFormatter` handles presentation logic that needs both visual verification and compatibility testing.

> **Decision: Component-First Testing Approach**
>
> - **Context**: Testing can proceed component-by-component or through integrated scenarios. Component-first testing isolates failures but may miss integration issues.
> - **Options Considered**: Component isolation, integration-first testing, hybrid approach
> - **Decision**: Start with component isolation, then build integration tests
> - **Rationale**: Dynamic programming algorithms are complex enough to warrant isolated verification before adding integration complexity. Individual component bugs are easier to diagnose and fix than integration failures.
> - **Consequences**: Enables parallel test development, simplifies debugging, but requires additional integration test layer

Testing Approach	Pros	Cons	Chosen?
Component Isolation	Clear failure attribution, parallel development, algorithmic focus	May miss integration issues, duplicate test data	✓ Primary
Integration-First	Tests real scenarios, catches interface mismatches early	Complex failure diagnosis, coupled test development	Secondary
Hybrid Approach	Balances isolation and integration, comprehensive coverage	Longer test development time, more complex test suite	Long-term goal

FileReader Component Testing

The `FileReader` component requires extensive testing of file I/O edge cases, encoding detection, and line normalization behavior. Test cases must cover different file encodings, line ending formats, and boundary

conditions that real-world files present.

Encoding Detection Test Matrix:

File Content	Expected Encoding	Test Purpose
Pure ASCII text	UTF-8	Basic encoding detection
UTF-8 with BOM	UTF-8	BOM handling verification
UTF-8 without BOM	UTF-8	Standard UTF-8 detection
Latin-1 characters (é, ñ, ü)	Latin-1	Fallback encoding detection
Binary data (null bytes)	Error	Binary file rejection
Empty file	UTF-8 (default)	Edge case handling
Mixed valid UTF-8 and invalid sequences	Error	Encoding corruption detection

Line Ending Normalization Test Cases:

The `normalize_line_endings` function must handle all combinations of line ending types while preserving the original format information for diff headers. Test cases verify that different line ending formats are correctly detected and normalized to a consistent internal representation.

Input Content	Expected Normalized	Expected Original Format	Test Purpose
`"line1\nline2\n"`	`["line1", "line2"]`	`LF`	Unix format handling
`"line1\r\nline2\r\n"`	`["line1", "line2"]`	`CRLF`	Windows format handling
`"line1\rline2\r"`	`["line1", "line2"]`	`CR`	Classic Mac format handling
`"line1\r\nline2\n"`	`["line1", "line2"]`	`"mixed"`	Mixed format detection
`"line1\nline2"`	`["line1", "line2"]`	`"no_final_newline"`	Missing final newline
`""`	`[]`	`"empty"`	Empty file handling
`"\n\n\n"`	`["", "", ""]`	`LF`	Multiple empty lines

File Content Creation Test Scenarios:

The `create_file_content` factory function requires validation of its input normalization and validation logic. Tests must verify that invalid combinations are rejected and valid inputs produce correctly structured `FileContent` objects.

```python
# Test case structure for FileContent creation

test_cases = [

    {

        'filepath': '/path/to/file.txt',

        'raw_content': 'line1\nline2\nline3',

        'encoding': 'UTF-8',

        'line_ending': 'LF',

        'expected_lines': ['line1', 'line2', 'line3'],

        'expected_line_count': 3

    },

    {

        'filepath': '/path/to/empty.txt',

        'raw_content': '',

        'encoding': 'UTF-8',

        'line_ending': 'empty',

        'expected_lines': [],

        'expected_line_count': 0

    }

]
```

LCS Engine Component Testing

The `LCSEngine` requires rigorous algorithmic testing since dynamic programming implementations are prone to off-by-one errors, incorrect recurrence relations, and backtracking mistakes. Testing approaches include mathematical verification against known solutions, property-based testing, and performance validation.

Known Solution Verification:

LCS algorithms can be verified against manually computed solutions for small inputs, then scaled to larger inputs with predictable patterns. The test suite includes carefully constructed examples where the correct LCS is obvious

and can be manually verified.

Sequence 1	Sequence 2	Expected LCS	Expected Length	Test Purpose
`["A", "B", "C"]`	`["A", "B", "C"]`	`["A", "B", "C"]`	3	Identical sequences
`["A", "B", "C"]`	`["X", "Y", "Z"]`	`[]`	0	No common elements
`["A", "B", "C", "D"]`	`["B", "D", "F"]`	`["B", "D"]`	2	Partial overlap
`[]`	`["A", "B"]`	`[]`	0	Empty first sequence
`["A", "B"]`	`[]`	`[]`	0	Empty second sequence
`[]`	`[]`	`[]`	0	Both sequences empty
`["A", "A", "A"]`	`["A", "A"]`	`["A", "A"]`	2	Repeated elements

Matrix Construction Verification:

The `build_lcs_matrix` function requires validation of its dynamic programming table construction. Tests verify that the matrix dimensions are correct and that each cell contains the expected LCS length for its corresponding subsequences.

```python
# Example matrix verification for sequences ["A", "B"] and ["A", "C"]
expected_matrix = [
    [0, 0, 0],  # Row 0: empty vs ["A", "C"]
    [0, 1, 1],  # Row 1: ["A"] vs ["A", "C"]
    [0, 1, 1]   # Row 2: ["A", "B"] vs ["A", "C"]
]
# Matrix[i][j] represents LCS length of first i elements vs first j elements
```

Backtracking Algorithm Verification:

The `backtrack` function must correctly reconstruct the actual LCS from the completed matrix. Tests verify that backtracking produces a valid LCS (present in both sequences in order) and that multiple valid LCS paths are handled consistently.

Matrix State	Expected Backtrack Path	Expected LCS	Test Purpose
Single optimal path	Deterministic trace	Unique LCS	Standard case
Multiple optimal paths	Consistent tie-breaking	One valid LCS	Tie-breaking rules
All zeros matrix	No backtrack moves	Empty LCS	No common elements
Diagonal matrix	Straight diagonal trace	Full common sequence	Identical inputs

Performance and Memory Testing:

The `LCSEngine` must handle large inputs without memory exhaustion while maintaining reasonable performance. Tests verify that the memory optimization strategies work correctly and that resource limits are respected.

Sequence Length	Expected Algorithm	Expected Memory	Expected Time	Test Purpose
100 x 100	`matrix`	~80KB	<1ms	Small input baseline
1000 x 1000	`matrix`	~8MB	<100ms	Medium input handling
10000 x 10000	`two_row`	~80KB	<10s	Memory optimization
50000 x 50000	`hirschberg`	~800KB	<5min	Large input handling

DiffGenerator Component Testing

The `DiffGenerator` transforms LCS results into structured diff operations and hunks. Testing focuses on correct edit operation generation, proper hunk formation with context lines, and unified diff format compliance.

Edit Operation Generation Testing:

The `lcs_to_edit_operations` function must correctly identify which lines are unchanged (in LCS), added (in sequence2 but not LCS), or deleted (in sequence1 but not LCS). Tests verify that the generated operations correctly represent the transformation from sequence1 to sequence2.

File1 Lines	File2 Lines	LCS	Expected Operations	Test Purpose
`["A", "B", "C"]`	`["A", "B", "C"]`	`["A", "B", "C"]`	All `UNCHANGED`	Identical files
`["A", "B"]`	`["A", "X", "B"]`	`["A", "B"]`	`UNCHANGED`, `ADDED`, `UNCHANGED`	Single insertion
`["A", "X", "B"]`	`["A", "B"]`	`["A", "B"]`	`UNCHANGED`, `DELETED`, `UNCHANGED`	Single deletion
`["A", "B"]`	`["X", "Y"]`	`[]`	`DELETED`, `DELETED`, `ADDED`, `ADDED`	Complete replacement

Hunk Formation Testing:

The `diff_lines_to_hunks` function groups nearby changes into hunks with appropriate context lines. Tests verify that hunks are formed correctly, context lines are included properly, and adjacent hunks are merged when appropriate.

```python
# Test case: Context line grouping
diff_lines = [
    DiffLine("line1", UNCHANGED, 1, 1),
    DiffLine("line2", UNCHANGED, 2, 2),
    DiffLine("old_line", DELETED, 3, None),
    DiffLine("new_line", ADDED, None, 3),
    DiffLine("line4", UNCHANGED, 4, 4),
    DiffLine("line5", UNCHANGED, 5, 5)
]

# With context_lines=2, expect single hunk:
expected_hunk = Hunk(
    old_start=1, old_count=5,
    new_start=1, new_count=5,
    lines=diff_lines,
    context_before=2,
    context_after=2
)
```

Hunk Merging Logic Testing:

The `merge_adjacent_hunks` function combines hunks that are close enough together to warrant merging. Tests verify the merging distance calculation and ensure that merged hunks maintain correct line counts and ranges.

Hunk1 Range	Hunk2 Range	Context Lines	Max Gap	Should Merge?	Test Purpose
Lines 1-5	Lines 8-12	2	3	Yes	Close hunks merge
Lines 1-5	Lines 15-20	2	3	No	Distant hunks separate
Lines 1-5	Lines 6-10	1	0	Yes	Adjacent hunks merge
Lines 1-5	Lines 7-10	0	1	No	Context gap prevents merge

OutputFormatter Component Testing

The `OutputFormatter` handles presentation logic including unified diff format generation, ANSI color application, and CLI integration. Testing verifies format compliance, color handling, and cross-platform compatibility.

Unified Diff Format Compliance:

The `format_unified_diff` function must generate output that complies with the standard unified diff format. Tests verify header generation, hunk marker format, line prefix correctness, and overall format structure.

```python
# Expected unified diff format structure
expected_format = [
    "--- file1.txt\toriginal_timestamp",
    "+++ file2.txt\tmodified_timestamp",
    "@@ -1,3 +1,3 @@",
    " unchanged_line",
    "-deleted_line",
    "+added_line",
    " another_unchanged_line"
]
```

ANSI Color Testing:

The `ColorFormatter` must correctly apply ANSI color codes when appropriate and strip them when color is disabled. Tests verify color application, TTY detection, and cross-platform compatibility.

Input Line	Color Enabled	Expected Output	Test Purpose
`DiffLine("text", DELETED, 1, None)`	True	`"\033[31m-text\033[0m"`	Red deletion color
`DiffLine("text", ADDED, None, 1)`	True	`"\033[32m+text\033[0m"`	Green addition color
`DiffLine("text", UNCHANGED, 1, 1)`	True	`" text"`	No color for unchanged
`DiffLine("text", DELETED, 1, None)`	False	`"-text"`	No color when disabled

CLI Argument Processing Testing:

The argument parsing logic must correctly handle all supported command-line options and provide appropriate error messages for invalid usage. Tests verify option parsing, file path validation, and help text generation.

```python
# CLI argument test cases
cli_test_cases = [
    {
        'args': ['file1.txt', 'file2.txt'],
        'expected': DiffArguments(file1='file1.txt', file2='file2.txt',
                                  color=True, context_lines=3),
        'description': 'Basic two-file comparison'
    },
    {
        'args': ['--no-color', 'file1.txt', 'file2.txt'],
        'expected': DiffArguments(file1='file1.txt', file2='file2.txt',
                                  color=False, context_lines=3),
        'description': 'Color disabled'
    },
    {
        'args': ['--context', '5', 'file1.txt', 'file2.txt'],
        'expected': DiffArguments(file1='file1.txt', file2='file2.txt',
                                  color=True, context_lines=5),
        'description': 'Custom context lines'
    }
]
```

Property-Based Testing Strategy

Property-based testing generates random inputs and verifies that certain properties always hold, regardless of the specific input values. This approach is particularly valuable for diff algorithms because it can discover edge cases that manual test case construction might miss.

LCS Properties to Verify:

1. **Symmetry Property**: The LCS length of (A, B) equals the LCS length of (B, A)
2. **Subsequence Property**: The returned LCS must be a valid subsequence of both input sequences
3. **Optimality Property**: No longer common subsequence should exist than the one returned

4. **Prefix Property**: LCS(A[0:i], B[0:j]) should be consistent with the full LCS computation

Diff Generation Properties:

1. **Roundtrip Property**: Applying the generated diff operations should transform sequence1 into sequence2
2. **Line Conservation**: The total number of unchanged + deleted lines should equal the original file length
3. **Hunk Completeness**: All diff operations should appear in exactly one hunk
4. **Context Consistency**: Context lines in hunks should match the original file content

Error Condition Testing

Each component must handle error conditions gracefully and provide meaningful error messages. Error testing verifies that components fail fast with appropriate error types and context information.

FileReader Error Scenarios:

Error Condition	Expected Exception	Expected Message Pattern	Recovery Strategy
File not found	`FileSystemError`	"File not found: {filepath}"	Prompt user for correct path
Permission denied	`FileSystemError`	"Permission denied: {filepath}"	Check file permissions
Binary file detection	`EncodingError`	"Binary file detected: {filepath}"	Skip or warn user
Mixed line endings	Warning	"Mixed line endings detected"	Continue with normalization
Encoding detection failure	`EncodingError`	"Could not determine encoding"	Prompt user for encoding

LCSEngine Error Scenarios:

Error Condition	Expected Exception	Expected Message Pattern	Recovery Strategy
Memory limit exceeded	`ResourceError`	"LCS matrix too large: {size}MB > {limit}MB"	Suggest memory optimization
Time limit exceeded	`ResourceError`	"LCS computation timeout after {seconds}s"	Suggest algorithm optimization
Invalid sequence input	`AlgorithmError`	"Invalid sequence type: expected List[str]"	Validate input format
Empty matrix allocation	`ResourceError`	"Cannot allocate matrix of size {rows}x{cols}"	Check available memory

Common Pitfalls in Testing

Testing diff algorithms presents several common pitfalls that can lead to false confidence in correctness or missed edge cases. Understanding these pitfalls helps construct more robust test suites.

⚠️ Pitfall: Testing Only Happy Path Scenarios

Many developers focus testing on scenarios where files have clear, obvious differences and ignore the edge cases where algorithms are most likely to fail. Testing only well-formed text files with consistent line endings misses the real-world complexity of mixed encodings, binary data, and malformed input.

The fix involves systematically constructing adversarial test cases: files with no common lines, files where one is empty, files with only whitespace differences, and files with encoding issues. Create a test matrix that covers all combinations of edge conditions rather than just the scenarios that work smoothly.

⚠️ Pitfall: Assuming LCS Uniqueness

LCS algorithms can produce multiple valid results when there are multiple longest common subsequences of equal length. Tests that expect a specific LCS may fail when the algorithm chooses a different but equally valid path through the edit graph.

The solution is to test LCS properties rather than exact content. Verify that the returned LCS has the correct length, appears in both sequences in the correct order, and that no longer common subsequence exists. This allows the algorithm flexibility in tie-breaking while ensuring correctness.

⚠️ Pitfall: Off-by-One Line Number Testing

Unified diff format uses one-indexed line numbers, but most programming languages use zero-indexed arrays. Tests that don't account for this indexing difference may pass with incorrect line numbering that breaks compatibility with standard diff tools.

Create test cases that specifically verify line numbering in hunk headers matches the expected unified diff format. Compare generated output with the output of system diff tools to ensure compatibility. Pay special attention to edge cases like single-line files and changes at the beginning or end of files.

⚠️ Pitfall: Incomplete Context Line Testing

Context line generation involves complex boundary conditions when changes occur near file boundaries or when hunks would overlap. Tests that don't exercise these boundary conditions miss scenarios where context line calculation fails.

Construct test cases where changes occur on the first line, last line, and cases where requested context extends beyond file boundaries. Verify that hunk merging works correctly when context lines overlap and that merged hunks maintain correct line counts.

⚠️ Pitfall: Platform-Specific Color Testing

ANSI color code handling varies significantly between operating systems and terminal types. Tests that work on one platform may fail on others due to different color support, TTY detection, or output redirection behavior.

Create platform-specific test configurations that verify color output under different conditions: interactive terminal, redirected output, Windows command prompt, and Unix terminals. Mock TTY detection to ensure consistent behavior across environments.

Milestone Checkpoints

Each milestone in the diff tool implementation requires specific validation checkpoints that verify both individual component functionality and integration with previous components. These checkpoints provide concrete verification steps and expected outcomes.

Milestone 1 Checkpoint: Line Tokenization

After implementing the `FileReader` component, the following validation steps ensure correct file reading and line processing behavior:

Validation Commands:

```bash
# Basic functionality test
python -m pytest tests/test_file_reader.py -v

# Encoding detection test
python test_encoding_detection.py

# Line normalization verification
python test_line_endings.py
```

Expected Behavior Verification:

1. **File Reading Test**: Create test files in different encodings (UTF-8, Latin-1) with various content types. The `read_file_lines` function should correctly detect encoding and return a list of strings representing file lines.
2. **Line Ending Normalization**: Create files with different line endings (LF, CRLF, CR, mixed) and verify that `normalize_line_endings` correctly detects the original format while producing consistent internal representation.
3. **Empty Line Preservation**: Files with empty lines should preserve those empty lines in the output. The line count should match the actual number of lines including empty ones.
4. **Error Handling Verification**: Binary files should be rejected with appropriate error messages. Files with permission issues should produce clear error descriptions.

Success Indicators:

- All encoding detection tests pass with correct UTF-8/Latin-1 identification
- Line count reported matches manual line count for various file types
- Empty lines are preserved in output without being filtered
- Binary file detection prevents processing of non-text files
- Error messages provide actionable information for file access issues

Troubleshooting Common Issues:

Symptom	Likely Cause	Diagnostic Steps	Fix
Binary files accepted	Missing binary detection	Check for null bytes in content	Add binary detection logic
Line count mismatch	Trailing newline handling	Compare with `wc -l` output	Handle final newline consistently
Encoding errors	Detection order wrong	Test with Latin-1 content	Try UTF-8 first, fallback to Latin-1
Empty lines missing	Split logic incorrect	Check split algorithm	Use split that preserves empty strings

Milestone 2 Checkpoint: LCS Algorithm

The LCS implementation requires verification of correct dynamic programming matrix construction and backtracking logic:

Validation Commands:

```bash
# LCS algorithm correctness
python -m pytest tests/test_lcs_engine.py -v

# Performance and memory testing
python test_lcs_performance.py

# Matrix construction verification
python test_matrix_building.py
```

Expected Behavior Verification:

1. **Algorithm Correctness**: Test with known LCS examples where the result can be manually verified. The `compute_lcs` function should return the correct longest common subsequence and length.
2. **Matrix Construction**: The `build_lcs_matrix` function should produce a properly sized matrix where each cell contains the correct LCS length for the corresponding subsequences.
3. **Backtracking Verification**: The `backtrack` function should reconstruct a valid LCS that appears in both input sequences in the correct order.
4. **Memory Optimization**: Large inputs should trigger memory optimization strategies without affecting correctness.

Success Indicators:

- All known LCS test cases produce correct results
- Matrix dimensions match input sequence lengths plus one
- Backtracked LCS is a valid subsequence of both inputs
- Memory usage stays within configured limits for large inputs
- Performance scales reasonably with input size

Troubleshooting Common Issues:

Symptom	Likely Cause	Diagnostic Steps	Fix
Wrong LCS length	Matrix construction error	Print matrix values manually	Check recurrence relation
Backtrack fails	Index boundaries wrong	Trace backtrack path	Fix matrix indexing
Memory explosion	No optimization triggered	Check input size thresholds	Lower optimization trigger point
Performance issues	Inefficient matrix access	Profile algorithm execution	Optimize inner loop

Milestone 3 Checkpoint: Diff Generation

The diff generation component transforms LCS results into structured hunks with unified diff format:

Validation Commands:

```bash
# Diff generation correctness
python -m pytest tests/test_diff_generator.py -v

# Hunk formation testing
python test_hunk_generation.py

# Format compliance verification
python test_unified_diff_format.py
```

Expected Behavior Verification:

1. **Edit Operations**: The `lcs_to_edit_operations` function should correctly identify added, deleted, and unchanged lines based on LCS results.
2. **Hunk Formation**: The `diff_lines_to_hunks` function should group nearby changes with appropriate context lines and generate correct hunk boundaries.
3. **Line Numbering**: All line numbers in hunks should be one-indexed and accurately reflect positions in the original files.
4. **Context Handling**: Context lines should be included around changes, and overlapping contexts should trigger hunk merging.

Success Indicators:

- Edit operations correctly represent file transformation
- Hunk line counts match actual line content
- Line numbering follows unified diff conventions (one-indexed)
- Context lines accurately match original file content
- Adjacent hunks merge appropriately based on context overlap

Troubleshooting Common Issues:

Symptom	Likely Cause	Diagnostic Steps	Fix
Wrong line numbers	Zero-based indexing used	Check hunk headers against file	Convert to one-based numbering
Missing context lines	Context calculation error	Verify context boundaries	Fix context line extraction
Hunks not merging	Merge distance wrong	Check overlap calculation	Adjust merge threshold
Line count mismatch	Hunk calculation error	Sum hunk lines vs file lines	Fix hunk line counting

Milestone 4 Checkpoint: CLI and Color Output

The final milestone integrates all components into a complete CLI tool with colored output:

Validation Commands:

```bash
# Complete integration test
python diff_tool.py file1.txt file2.txt

# Color output verification
python diff_tool.py --color file1.txt file2.txt

# Context line configuration
python diff_tool.py --context 5 file1.txt file2.txt

# No-color mode testing
python diff_tool.py --no-color file1.txt file2.txt > output.diff
```

Expected Behavior Verification:

1. **CLI Integration**: Command-line arguments should be parsed correctly with appropriate defaults and error messages for invalid usage.
2. **Color Output**: Terminal output should include ANSI color codes for additions (green) and deletions (red) when appropriate, but strip colors when output is redirected.
3. **Format Compatibility**: Generated diff output should be compatible with standard diff tools and patch utilities.
4. **Exit Codes**: Program should return 0 for identical files, 1 for different files, and 2 for errors.

Success Indicators:

- CLI accepts all documented arguments correctly
- Color output appears properly in terminal
- No-color mode produces clean output suitable for redirection
- Generated diffs can be applied using standard patch tools
- Exit codes match diff tool conventions
- Error messages provide helpful guidance for incorrect usage

Integration Verification Steps:

1. **Compare with System Diff**: Run both your tool and system diff on the same files and verify that the essential differences are captured correctly (line numbers and change indicators may vary slightly due to different algorithms).
2. **Patch Application Test**: Generate a diff with your tool and verify it can be applied using standard patch utilities to reproduce the target file.
3. **Performance Validation**: Verify that the complete pipeline handles reasonably sized files (1000+ lines) within acceptable time limits.
4. **Cross-Platform Testing**: Test CLI behavior on different operating systems to ensure consistent argument parsing and color output.

Final Integration Troubleshooting:

Symptom	Likely Cause	Diagnostic Steps	Fix
Colors in redirected output	TTY detection failed	Test with `diff_tool.py > file`	Fix TTY detection logic
Patch application fails	Format compliance issue	Compare with system diff output	Align format with standards
Wrong exit codes	Exit code logic missing	Test with identical/different files	Implement proper exit codes
Performance degradation	Component integration overhead	Profile pipeline execution	Optimize data passing between components

Implementation Guidance

The testing implementation focuses on creating a comprehensive test suite that validates each component individually and verifies their integration. The approach emphasizes systematic test case construction, property-based testing, and milestone-driven validation.

Technology Recommendations

Testing Component	Simple Option	Advanced Option
Test Framework	`unittest` (built-in)	`pytest` with fixtures
Property Testing	Manual test case construction	`hypothesis` for property-based testing
Performance Testing	Basic timing with `time.time()`	`pytest-benchmark` with statistical analysis
Test Data Management	Hardcoded test strings	External test files with various encodings
Coverage Analysis	Manual verification	`coverage.py` with branch coverage
CLI Testing	Direct function calls	`click.testing.CliRunner` for CLI simulation

Recommended File Structure

```
project-root/
├── src/
│   ├── diff_tool/
│   │   ├── __init__.py
│   │   ├── file_reader.py      ← FileReader component
│   │   ├── lcs_engine.py       ← LCSEngine component
│   │   ├── diff_generator.py   ← DiffGenerator component
│   │   ├── output_formatter.py ← OutputFormatter component
│   │   └── main.py             ← CLI entry point
├── tests/
│   ├── __init__.py
│   ├── conftest.py             ← pytest fixtures and shared test utilities
│   ├── test_file_reader.py     ← FileReader unit tests
│   ├── test_lcs_engine.py      ← LCSEngine unit tests
│   ├── test_diff_generator.py ← DiffGenerator unit tests
│   ├── test_output_formatter.py ← OutputFormatter unit tests
│   ├── test_integration.py     ← End-to-end integration tests
│   ├── test_properties.py      ← Property-based testing
│   ├── test_data/             ← Test files with various encodings
│   │   ├── utf8_sample.txt
│   │   ├── latin1_sample.txt
│   │   ├── binary_sample.bin
│   │   ├── empty_file.txt
│   │   └── mixed_endings.txt
│   └── performance/
│       ├── test_lcs_performance.py
│       └── test_memory_usage.py
├── pytest.ini                ← pytest configuration
└── requirements-test.txt      ← testing dependencies
```

Test Fixture Infrastructure

Create reusable test fixtures that provide consistent test data and component instances across the test suite:

PYTHON

```python
# tests/conftest.py - Shared test fixtures and utilities

import pytest

from pathlib import Path

from src.diff_tool.file_reader import FileReader

from src.diff_tool.lcs_engine import LCSEngine

from src.diff_tool.diff_generator import DiffGenerator

from src.diff_tool.output_formatter import OutputFormatter


@pytest.fixture

def file_reader():

    """Provide a configured FileReader instance for testing."""

    return FileReader()


@pytest.fixture

def lcs_engine():

    """Provide a configured LCSEngine instance for testing."""

    return LCSEngine()


@pytest.fixture

def diff_generator():

    """Provide a configured DiffGenerator instance for testing."""

    return DiffGenerator()


@pytest.fixture

def output_formatter():

    """Provide a configured OutputFormatter instance for testing."""

    return OutputFormatter()


@pytest.fixture

def test_data_dir():
```

```
    """Provide path to test data directory."""
    return Path(__file__).parent / "test_data"


@pytest.fixture
def sample_files(test_data_dir, tmp_path):
    """Create sample test files with known content and encoding."""
    # TODO: Create UTF-8 file with known content
    # TODO: Create Latin-1 file with accented characters
    # TODO: Create file with mixed line endings
    # TODO: Create empty file
    # TODO: Return dict mapping file types to file paths
    pass


@pytest.fixture
def known_lcs_cases():
    """Provide test cases with known LCS solutions."""
    return [
        {
            'seq1': ["A", "B", "C", "D"],
            'seq2': ["A", "X", "B", "Y", "C"],
            'expected_lcs': ["A", "B", "C"],
            'expected_length': 3
        },
        # TODO: Add more known LCS test cases
        # TODO: Include edge cases: empty sequences, no common elements
        # TODO: Include cases with repeated elements
    ]
```

Component Test Skeletons

Provide test class structures for each component with TODO comments mapping to specific test requirements:

PYTHON

```python
# tests/test_file_reader.py - FileReader component tests

import pytest

from src.diff_tool.file_reader import FileReader, FileContent

from src.diff_tool.errors import FileSystemError, EncodingError


class TestFileReader:

    """Test FileReader component functionality."""


    def test_detect_file_encoding_utf8(self, test_data_dir):

        """Test UTF-8 encoding detection for files with UTF-8 content."""

        # TODO: Create test file with UTF-8 content

        # TODO: Call detect_file_encoding on test file

        # TODO: Assert encoding is detected as 'UTF-8'

        pass


    def test_detect_file_encoding_latin1(self, test_data_dir):

        """Test Latin-1 encoding detection for files with Latin-1 content."""

        # TODO: Create test file with Latin-1 content (accented characters)

        # TODO: Call detect_file_encoding on test file

        # TODO: Assert encoding is detected as 'Latin-1'

        pass


    def test_read_file_lines_basic(self, sample_files, file_reader):

        """Test basic file reading with line splitting."""

        # TODO: Use sample UTF-8 file from fixture

        # TODO: Call read_file_lines to get FileContent object

        # TODO: Assert lines list matches expected content

        # TODO: Assert line_count matches len(lines)
```

```
        # TODO: Assert encoding is correctly detected
        pass


    def test_normalize_line_endings_unix(self, file_reader):
        """Test normalization of Unix LF line endings."""
        content = "line1\nline2\nline3\n"
        # TODO: Call normalize_line_endings on content
        # TODO: Assert normalized lines are ["line1", "line2", "line3"]
        # TODO: Assert original_endings is "LF"
        pass


    def test_normalize_line_endings_windows(self, file_reader):
        """Test normalization of Windows CRLF line endings."""
        # TODO: Test content with \r\n line endings
        # TODO: Verify normalization produces correct line list
        # TODO: Verify original format detection
        pass


    def test_binary_file_detection(self, test_data_dir, file_reader):
        """Test that binary files are detected and rejected."""
        # TODO: Create binary file with null bytes
        # TODO: Verify EncodingError is raised when reading
        # TODO: Verify error message indicates binary file
        pass


    def test_file_not_found_error(self, file_reader):
        """Test appropriate error for nonexistent files."""
        # TODO: Call read_file_lines on nonexistent file path
```

```
        # TODO: Assert FileSystemError is raised
        # TODO: Verify error message contains file path
        pass

# tests/test_lcs_engine.py - LCSEngine component tests
class TestLCSEngine:
    """Test LCSEngine dynamic programming implementation."""

    def test_compute_lcs_identical_sequences(self, lcs_engine):
        """Test LCS computation for identical sequences."""
        seq1 = ["A", "B", "C"]
        seq2 = ["A", "B", "C"]
        # TODO: Call compute_lcs on identical sequences
        # TODO: Assert returned LCS equals input sequence
        # TODO: Assert LCS length equals sequence length
        pass

    def test_compute_lcs_no_common_elements(self, lcs_engine):
        """Test LCS computation when sequences share no elements."""
        # TODO: Create sequences with no common elements
        # TODO: Verify LCS is empty list
        # TODO: Verify LCS length is 0
        pass

    def test_build_lcs_matrix_dimensions(self, lcs_engine):
        """Test that LCS matrix has correct dimensions."""
        seq1 = ["A", "B"]
        seq2 = ["X", "Y", "Z"]
```

```
        # TODO: Call build_lcs_matrix on test sequences
        # TODO: Assert matrix has (len(seq1) + 1) rows
        # TODO: Assert matrix has (len(seq2) + 1) columns
        # TODO: Verify matrix[0][j] == 0 for all j (empty sequence base case)
        # TODO: Verify matrix[i][0] == 0 for all i (empty sequence base case)
        pass

    def test_backtrack_simple_case(self, lcs_engine, known_lcs_cases):
        """Test backtracking for known LCS cases."""
        for test_case in known_lcs_cases:
            # TODO: Build LCS matrix for test case sequences
            # TODO: Call backtrack on completed matrix
            # TODO: Assert backtracked LCS matches expected result
            # TODO: Verify LCS is valid subsequence of both inputs
        pass

    def test_memory_optimization_trigger(self, lcs_engine):
        """Test that memory optimization triggers for large inputs."""
        # TODO: Create large test sequences that exceed memory threshold
        # TODO: Mock memory monitoring to verify optimization triggered
        # TODO: Verify algorithm switches to memory-efficient version
        # TODO: Ensure result correctness is maintained
        pass

# tests/test_diff_generator.py - DiffGenerator component tests
class TestDiffGenerator:
    """Test DiffGenerator hunk formation and edit operations."""
```

```python
    def test_lcs_to_edit_operations_basic(self, diff_generator):
        """Test conversion of LCS to basic edit operations."""
        file1_lines = ["A", "B", "C"]
        file2_lines = ["A", "X", "C"]
        lcs = ["A", "C"]
        # TODO: Call lcs_to_edit_operations with test data
        # TODO: Assert operations are [UNCHANGED, DELETED, ADDED, UNCHANGED]
        # TODO: Verify line numbers are correctly assigned
        # TODO: Check that content matches source lines
        pass


    def test_diff_lines_to_hunks_single_change(self, diff_generator):
        """Test hunk formation for single isolated change."""
        # TODO: Create DiffLine list with single change surrounded by unchanged lines
        # TODO: Call diff_lines_to_hunks with context_lines=2
        # TODO: Assert single hunk is created
        # TODO: Verify hunk contains appropriate context lines
        # TODO: Check hunk line counts and ranges
        pass


    def test_merge_adjacent_hunks(self, diff_generator):
        """Test merging of hunks that are close together."""
        # TODO: Create two hunks with small gap between them
        # TODO: Call merge_adjacent_hunks with appropriate max_gap
        # TODO: Assert hunks are merged into single hunk
        # TODO: Verify merged hunk has correct line counts
        # TODO: Check that all lines from both hunks are preserved
        pass
```

```
    def test_hunk_line_numbering(self, diff_generator):
        """Test that hunk line numbers follow one-indexed conventions."""
        # TODO: Create test diff operations with known line positions
        # TODO: Generate hunks from operations
        # TODO: Verify hunk headers use one-based line numbering
        # TODO: Check old_start, old_count, new_start, new_count values
        pass

# tests/test_output_formatter.py - OutputFormatter component tests
class TestOutputFormatter:
    """Test OutputFormatter unified diff generation and color output."""

    def test_format_unified_diff_headers(self, output_formatter):
        """Test generation of unified diff file headers."""
        file1 = FileContent("file1.txt", ["line1"], 1, "UTF-8", "LF")
        file2 = FileContent("file2.txt", ["line2"], 1, "UTF-8", "LF")
        hunks = []
        # TODO: Call format_unified_diff to generate output
        # TODO: Assert output starts with "--- file1.txt" header
        # TODO: Assert second line is "+++ file2.txt" header
        # TODO: Verify header format matches unified diff standard
        pass

    def test_ansi_color_formatting(self, output_formatter):
        """Test ANSI color code application."""
        # TODO: Create DiffLine with DELETED type
        # TODO: Format line with colors enabled
```

```
        # TODO: Assert output contains red ANSI codes
        # TODO: Test ADDED lines get green colors
        # TODO: Verify UNCHANGED lines have no color
        pass

    def test_no_color_mode(self, output_formatter):
        """Test plain text output when colors are disabled."""
        # TODO: Configure formatter with colors disabled
        # TODO: Format diff lines of various types
        # TODO: Assert no ANSI codes appear in output
        # TODO: Verify line prefixes are still correct
        pass

# tests/test_integration.py - End-to-end integration tests
class TestIntegration:
    """Test complete diff tool pipeline integration."""

    def test_complete_diff_pipeline(self, sample_files):
        """Test full pipeline from file reading to diff output."""
        # TODO: Use sample files with known differences
        # TODO: Run complete diff pipeline: read -> LCS -> generate -> format
        # TODO: Verify output contains expected diff structure
        # TODO: Check that generated diff could be applied with patch
        pass

    def test_identical_files_handling(self, sample_files):
        """Test behavior when comparing identical files."""
        # TODO: Compare file with itself
```

```
        # TODO: Verify no hunks are generated
        # TODO: Check appropriate exit code (0 for identical)
        pass


    def test_performance_large_files(self, tmp_path):
        """Test performance with reasonably large files."""
        # TODO: Generate large test files (1000+ lines)
        # TODO: Run diff pipeline with timing
        # TODO: Verify completion within reasonable time limit
        # TODO: Check memory usage stays within bounds
        pass
```

Property-Based Test Implementation

Property-based testing automatically generates test inputs and verifies that important properties hold across all generated cases:

PYTHON

```python
# tests/test_properties.py - Property-based testing

from hypothesis import given, strategies as st

import pytest

from src.diff_tool.lcs_engine import LCSEngine

from src.diff_tool.diff_generator import DiffGenerator


class TestLCSProperties:

    """Property-based tests for LCS algorithm correctness."""



    @given(st.lists(st.text(min_size=1, max_size=10), max_size=20))

    def test_lcs_subsequence_property(self, sequence):

        """Test that LCS result is valid subsequence of both inputs."""

        lcs_engine = LCSEngine()

        # TODO: Generate second sequence as permutation/subset of first

        # TODO: Compute LCS of both sequences

        # TODO: Verify LCS appears in both sequences in correct order

        # TODO: Assert LCS length <= min(len(seq1), len(seq2))

        pass



    @given(st.lists(st.text(), max_size=15), st.lists(st.text(), max_size=15))

    def test_lcs_symmetry_property(self, seq1, seq2):

        """Test that LCS(A,B) has same length as LCS(B,A)."""

        lcs_engine = LCSEngine()

        # TODO: Compute LCS length for (seq1, seq2)

        # TODO: Compute LCS length for (seq2, seq1)

        # TODO: Assert both lengths are equal

        pass


class TestDiffProperties:
```

```
    """Property-based tests for diff generation correctness."""


    @given(st.lists(st.text(), max_size=20), st.lists(st.text(), max_size=20))
    def test_diff_roundtrip_property(self, file1_lines, file2_lines):
        """Test that applying diff operations transforms file1 to file2."""
        # TODO: Compute LCS of input files
        # TODO: Generate edit operations from LCS
        # TODO: Apply operations to file1_lines
        # TODO: Assert result equals file2_lines
        pass
```

Milestone Validation Scripts

Create standalone validation scripts that can be run after completing each milestone:

PYTHON

```python
# scripts/validate_milestone_1.py - File reading validation

#!/usr/bin/env python3

"""Validate Milestone 1: Line Tokenization implementation."""


def validate_file_reading():

    """Validate basic file reading functionality."""

    print("=== Milestone 1 Validation: Line Tokenization ===")



    # TODO: Test encoding detection with sample files

    # TODO: Verify line normalization with different endings

    # TODO: Check empty line preservation

    # TODO: Test binary file rejection

    # TODO: Report validation results with pass/fail status



    print("Validation complete. Check results above.")


if __name__ == "__main__":

    validate_file_reading()


# scripts/validate_milestone_2.py - LCS algorithm validation

#!/usr/bin/env python3

"""Validate Milestone 2: LCS Algorithm implementation."""


def validate_lcs_computation():

    """Validate LCS dynamic programming correctness."""

    print("=== Milestone 2 Validation: LCS Algorithm ===")



    # TODO: Test with known LCS examples

    # TODO: Verify matrix construction correctness

    # TODO: Check backtracking produces valid LCS
```

```
    # TODO: Test memory optimization for large inputs
    # TODO: Report algorithm performance metrics

    print("Validation complete. Check results above.")

if __name__ == "__main__":
    validate_lcs_computation()
```

Debugging and Performance Analysis Tools

Create utilities to help diagnose issues and analyze performance during development:

PYTHON

```python
# tools/debug_lcs.py - LCS algorithm debugging utilities

def visualize_lcs_matrix(seq1, seq2, matrix):

    """Print LCS matrix in readable format for debugging."""

    # TODO: Print sequence labels across top and side

    # TODO: Display matrix values in aligned grid

    # TODO: Highlight backtrack path if provided

    # TODO: Show final LCS result below matrix

    pass


def trace_backtrack_path(matrix, seq1, seq2):

    """Show step-by-step backtracking process."""

    # TODO: Start from bottom-right matrix cell

    # TODO: Print each backtrack decision and reasoning

    # TODO: Show how LCS is built in reverse

    # TODO: Display final LCS reconstruction

    pass


# tools/performance_profiler.py - Performance analysis utilities

def profile_lcs_performance(sequence_lengths):

    """Profile LCS performance across different input sizes."""

    # TODO: Generate test sequences of specified lengths

    # TODO: Time LCS computation for each size

    # TODO: Measure memory usage during computation

    # TODO: Plot performance curves

    # TODO: Identify performance bottlenecks

    pass


def memory_usage_analyzer():

    """Analyze memory usage patterns during LCS computation."""
```

```
    # TODO: Monitor memory allocation during matrix building
    # TODO: Track peak memory usage
    # TODO: Verify memory optimization triggers correctly
    # TODO: Report memory efficiency metrics
    pass
```

Debugging Guide

> **Milestone(s):** All milestones — debugging skills are essential throughout line tokenization (Milestone 1), LCS computation (Milestone 2), diff generation (Milestone 3), and CLI output (Milestone 4), with algorithm-specific techniques for each stage

Mental Model: Medical Diagnosis for Algorithms — Debugging a diff tool is like diagnosing a patient with multiple interconnected systems. Each component (File Reader, LCS Engine, Diff Generator, Output Formatter) represents an organ system, and symptoms in one area often indicate problems elsewhere. Just as doctors use systematic diagnostic procedures, we need structured approaches to identify the root cause of algorithmic failures. The key is understanding which symptoms point to which underlying conditions, and having the right diagnostic tools to peer inside each "organ" of our diff algorithm.

Debugging diff algorithms presents unique challenges because failures can cascade through the pipeline in subtle ways. A file encoding issue in the File Reader can manifest as incorrect LCS computation, which then produces malformed diff hunks, ultimately resulting in garbled terminal output. The dynamic programming nature of LCS computation means that small indexing errors can propagate exponentially, while the contextual grouping in diff generation creates complex interdependencies between seemingly unrelated lines.

The debugging process requires both systematic symptom analysis and deep algorithmic understanding. Unlike simple business logic bugs, diff algorithm failures often involve mathematical properties like optimal substructure, correctness of recurrence relations, and invariant preservation across component boundaries. This section provides learners with diagnostic frameworks specifically designed for these algorithmic challenges.

Symptom-Cause-Fix Reference

The following comprehensive reference table maps common symptoms encountered during diff tool implementation to their likely root causes and specific remediation strategies. Each entry includes detection techniques and step-by-step resolution procedures.

Symptom	Likely Cause	Detection Technique	Specific Fix
Matrix index out of bounds during LCS computation	Off-by-one error in matrix dimensions or loop bounds	Add boundary checks: `if i < 0 or i >= len(matrix)` before every access	Matrix should be `(len(seq1)+1) x (len(seq2)+1)`. Loop indices should use `range(1, len(seq1)+1)` and `range(1, len(seq2)+1)` for filling
LCS returns empty result for files with obvious common lines	Backtracking algorithm moving in wrong direction through matrix	Print backtrack path: log each `(i,j)` position and movement decision	Fix backtracking conditions: move diagonally when `seq1[i-1] == seq2[j-1]`, not when matrix values are equal
Memory error or system freeze during LCS on large files	Algorithm using O(mn) space without optimization	Monitor memory usage: `psutil.Process().memory_info().rss` during computation	Implement two-row optimization: only keep current and previous matrix rows, or switch to Hirschberg's algorithm
Diff output shows wrong line numbers in hunk headers	Confusion between 0-indexed internal arrays and 1-indexed diff format	Verify: print internal line numbers vs diff output line numbers side by side	Add 1 to all line numbers when formatting: `old_line_num + 1` and `new_line_num + 1` in hunk headers
Missing lines or duplicated content in diff hunks	Incorrect edit operation assignment from LCS result	Trace edit script generation: log every ADD/DELETE/UNCHANGED decision with line content	Fix `lcs_to_edit_operations`: ensure each line from both files appears exactly once in edit operations
Hunk context lines show incorrect content	Context extraction using wrong line indices or ranges	Print context ranges: log `start_idx` and `end_idx` for each context region	Context should use original file line indices, not edit operation indices. Verify bounds checking against `len(file_lines)`

Symptom	Likely Cause	Detection Technique	Specific Fix
File reading fails with encoding errors	Binary file processed as text, or wrong encoding assumption	Check file type: `file --mime-type filename` and inspect first 100 bytes as hex	Implement binary detection: if file contains null bytes or high percentage of non-printable characters, reject as binary
Line endings inconsistent in output	Mixed line ending normalization or preservation failure	Hexdump file endings: `od -c filename	tail` to see actual line terminators
Empty files cause crashes	Missing edge case handling for zero-length sequences	Test with: `touch empty.txt && diff_tool empty.txt nonempty.txt`	Add checks: if `len(sequence) == 0`, return appropriate empty results instead of processing through normal algorithm
Identical files show differences	Whitespace normalization inconsistency or trailing newline handling	Compare raw bytes: `cmp file1 file2` vs your tool's result	Ensure identical normalization: both files must undergo same whitespace trimming and line ending conversion
Color codes appear in redirected output	TTY detection failure or missing --no-color support	Test: `diff_tool file1 file2 > output.txt && cat output.txt`	Fix TTY detection: use `sys.stdout.isatty()` and respect `--no-color` flag to disable ANSI codes
Hunks merge incorrectly or stay separate unexpectedly	Hunk merging logic using wrong distance calculation	Print hunk gaps: log line numbers between consecutive hunks and merge decisions	Gap calculation should be `hunk2.old_start - (hunk1.old_start + hunk1.old_count + hunk1.context_after)`. Merge if gap ≤ 2 * context_lines
Program hangs during file processing	Infinite loop in backtracking or file	Add progress logging: print percentage completion every 1000 iterations	Check loop termination: backtracking should always decrease `i` or `j`, file reading

Symptom	Likely Cause	Detection Technique	Specific Fix
	reading without progress		should check for EOF after each read operation
Diff output missing @@ hunk headers	Hunk header generation skipped or formatted incorrectly	Verify header format: should match regex `@@\s-\d+,\d+\s\+\d+,\d+\s@@`	Header format: `@@ -{old_start},{old_count} +{new_start},{new_count} @@` with 1-indexed line numbers
Exit code always 0 regardless of differences	Exit code logic not implemented or wrong conditions	Test: `diff_tool identical.txt identical.txt; echo $?` should be 0, different files should be 1	Set exit code based on diff result: `sys.exit(0)` if no differences found, `sys.exit(1)` if differences exist
Context lines duplicated at hunk boundaries	Overlapping context regions not handled during hunk merging	Check context overlap: log context ranges when merging adjacent hunks	When merging hunks, deduplicate overlapping context: shared lines should appear once, not in both hunks

⚠️ **Pitfall: Debugging with Print Statements** Many learners add `print()` statements throughout their algorithm but this creates several problems. Print debugging can mask timing issues, produce overwhelming output that obscures the actual problem, and interfere with proper output formatting (especially with color codes). Instead, use structured logging with levels (`logging.debug()`, `logging.info()`) and enable debug output only when needed via command-line flags.

⚠️ **Pitfall: Testing Only with Small Files** Algorithms that work perfectly on 10-line files can fail catastrophically on 1000-line files due to memory exhaustion, indexing errors that only manifest at scale, or performance degradation that reveals algorithmic flaws. Always test with files of varying sizes: empty files, single-line files, files with thousands of lines, and files with very long individual lines.

Debugging Techniques

The following systematic approaches provide structured methods for diagnosing and resolving issues specific to diff algorithm implementation. Each technique targets different aspects of the algorithm pipeline and provides concrete steps for investigation.

Matrix Inspection and Visualization

The LCS dynamic programming matrix contains the complete computational history and can reveal algorithmic errors that are invisible from final output alone. Understanding how to read and interpret the matrix is crucial for

debugging the core algorithm.

Matrix Validation Technique:

Create a matrix visualization function that displays the LCS matrix with sequence elements as headers. For sequences `["a", "b", "c"]` and `["a", "c"]`, the correct matrix should show:

```
    ""  a   c
""   0  0   0
a    0  1   1
b    0  1   1
c    0  1   2
```

The matrix reveals several key properties that must hold for correct LCS computation. Each cell `(i,j)` represents the LCS length for the first `i` elements of sequence 1 and first `j` elements of sequence 2. Values should never decrease when moving right or down through the matrix. The bottom-right cell contains the final LCS length. Diagonal moves (when characters match) should increment the value from the diagonal predecessor.

Backtracking Path Verification:

Implement a path tracer that logs each backtracking decision with its rationale. The backtracking algorithm should follow these decision rules consistently:

1. If characters match (`seq1[i-1] == seq2[j-1]`), move diagonally and include the character in LCS
2. If `matrix[i-1][j] > matrix[i][j-1]` , move up (deletion from sequence 1)
3. Otherwise, move left (insertion into sequence 1)

Log each decision: `"At (3,2): seq1[2]='c' != seq2[1]='c' - MISMATCH. matrix[2][2]=1, matrix[3][1]=1. Moving left."` This reveals logic errors in the backtracking conditions.

Common Matrix Debugging Patterns:

Values that are too high indicate incorrect recurrence relation implementation - check that matches only increment by 1 and non-matches take the maximum of adjacent cells. Zeros appearing in unexpected locations suggest initialization problems or boundary condition errors. A matrix filled entirely with 1s typically indicates the loop is only comparing the first characters repeatedly rather than iterating through sequences correctly.

Algorithm Execution Tracing

Step-by-step execution tracing reveals the decision-making process within each algorithm component and helps identify where expectations diverge from reality.

LCS Computation Tracing:

Implement detailed logging for each matrix cell computation. For each `(i,j)` position, log the comparison being made, the predecessor values consulted, and the resulting cell value. This creates an audit trail of the dynamic programming process:

```
Computing cell (2,3): comparing seq1[1]='b' with seq2[2]='c'
  Characters don't match
  Left neighbor matrix[2][2] = 1
  Top neighbor matrix[1][3] = 1
  Taking max(1,1) = 1
  Setting matrix[2][3] = 1
```

This granular tracing immediately reveals whether the algorithm is accessing correct sequence positions, using proper indexing, and applying the recurrence relation correctly.

Edit Operation Generation Tracing:

When converting LCS results to edit operations, trace each decision point that determines whether a line should be marked as UNCHANGED, ADDED, or DELETED. Log the current positions in both sequences, the characters being compared, and the reasoning for the operation type assignment:

```
LCS position 5: seq1[7]='def' matches seq2[12]='def' → UNCHANGED
Between matches: seq1[8]='ghi' not in LCS → DELETED
Next LCS match at seq2[13]='jkl', gap seq2[13]='extra' → ADDED
```

This reveals gaps in LCS coverage and ensures every line from both input sequences is properly classified in the edit script.

Hunk Formation Tracing:

The process of grouping edit operations into hunks with context involves complex boundary calculations and merging decisions. Trace each step of hunk formation:

```
Found change block: lines 10-15 (3 deletions, 2 additions)
Adding context: before=[7,8,9] after=[16,17,18]
Hunk created: old_start=8 old_count=9 new_start=8 new_count=8
Checking merge with next hunk at line 25: gap=7 > 2*context(3) → separate hunks
```

This debugging approach reveals incorrect line counting, context boundary errors, and merging logic failures.

Output Format Validation

Diff output must conform to standardized formats that other tools can parse correctly. Validation techniques ensure compliance with format specifications.

Unified Diff Format Compliance:

Implement a format validator that checks each element of the unified diff output against the specification. The validator should verify:

- File headers use exactly `---` and `+++` prefixes with proper spacing
- Hunk headers match the pattern `@@ -{old_start},{old_count} +{new_start},{new_count} @@`
- Line prefixes are exactly one character: space for unchanged, `-` for deleted, `+` for added
- Line numbers in headers are 1-indexed and match the actual line counts in the hunk body

- No trailing whitespace on empty lines (represented as single prefix character)

ANSI Color Code Validation:

When color output is enabled, verify that ANSI codes are properly paired and don't interfere with format parsing. Create a validator that strips ANSI codes and confirms the underlying text still matches unified diff format:

```python
def validate_color_output(colored_line):
    stripped = re.sub(r'\033\[[0-9;]*m', '', colored_line)
    return validate_diff_line_format(stripped)
```

Color codes should only wrap the content portion of diff lines, not the prefix characters or line numbers that tools use for parsing.

Line Number Consistency Checking:

Implement a checker that verifies line number consistency throughout the diff output. Track running totals of old and new line numbers as you process each hunk, ensuring that:

- Hunk header line counts match the actual number of lines in the hunk body
- Line number sequences are continuous with no gaps or duplicates
- UNCHANGED lines increment both old and new line counters
- DELETED lines increment only old line counter
- ADDED lines increment only new line counter

This catches off-by-one errors and incorrect line counting that can make diff output unusable by other tools.

Memory and Performance Debugging

Large file processing reveals memory leaks, inefficient algorithms, and resource management issues that don't appear with small test cases.

Memory Usage Monitoring:

Implement memory tracking that monitors heap usage throughout algorithm execution. Track peak memory usage during matrix construction, backtracking, and diff generation phases:

```python
import psutil
import gc

def track_memory(phase_name):
    gc.collect()  # Force garbage collection
    process = psutil.Process()
    memory_mb = process.memory_info().rss / 1024 / 1024
    print(f"{phase_name}: {memory_mb:.1f} MB")
```

Monitor for memory growth that doesn't match algorithmic expectations. The LCS matrix should require approximately `len(seq1) * len(seq2) * 8 bytes` for integer storage. Significantly higher usage indicates data structure inefficiency or memory leaks.

Algorithm Complexity Verification:

Measure actual runtime against theoretical complexity expectations. For sequences of length `m` and `n`, LCS computation should exhibit O(mn) time complexity. Test with progressively larger inputs and verify that runtime scales predictably:

- 100x100 sequences → baseline time T
- 200x200 sequences → approximately 4T
- 1000x1000 sequences → approximately 100T

Deviation from expected scaling indicates algorithmic inefficiency or implementation errors that cause redundant computation.

Resource Limit Testing:

Test algorithm behavior at system resource boundaries. Create test cases that approach memory limits, processing time limits, and file system constraints. This reveals whether the algorithm fails gracefully or crashes unpredictably:

```python
# Test with files approaching memory limits
large_file_size = psutil.virtual_memory().available // 2
create_test_file(large_file_size)

# Test with deep recursion (for recursive implementations)
sys.setrecursionlimit(100)  # Lower than default
test_with_long_sequences()
```

Property-Based Testing Integration

Property-based testing generates random inputs and verifies that algorithmic properties hold across diverse scenarios.

LCS Property Verification:

Implement property tests that verify fundamental LCS properties:

- **Subsequence Property**: Every element in the returned LCS must appear in both original sequences in the same relative order
- **Optimality Property**: No longer common subsequence should exist (verify by checking all possible extensions)
- **Symmetry Property**: `lcs(A, B)` should have the same length as `lcs(B, A)` (though elements may differ if multiple optimal solutions exist)

Edit Script Property Verification:

Verify that edit scripts correctly transform one sequence into another:

- **Transformation Property**: Applying all edit operations to sequence 1 should yield sequence 2
- **Minimality Property**: The total number of ADD and DELETE operations should equal the edit distance
- **Coverage Property**: Every line from both input sequences should appear exactly once across all edit operations

Round-Trip Testing:

Implement round-trip tests that verify the complete pipeline preserves information correctly:

1. Start with two known file contents
2. Process through complete diff pipeline
3. Apply the generated diff operations to reconstruct the second file
4. Verify the reconstructed content matches the original second file exactly

This end-to-end testing catches subtle bugs that unit tests might miss, especially issues involving line ending preservation, whitespace handling, and context line selection.

Implementation Guidance

Technology Recommendations

Component	Simple Option	Advanced Option
Logging Framework	Python `logging` with console handler	Structured logging with JSON output and log levels
Memory Profiling	`psutil` for basic memory monitoring	`memory_profiler` with line-by-line analysis
Test Generation	Manual test cases with known outputs	`hypothesis` for property-based test generation
Performance Measurement	`time.time()` for basic timing	`cProfile` with statistical analysis
Matrix Visualization	Print statements with formatted output	Rich terminal UI with color-coded matrices

Recommended File Structure

```
project-root/
  debug/
    matrix_visualizer.py     ← LCS matrix inspection tools
    trace_logger.py          ← Algorithm execution tracing
    property_tester.py       ← Property-based test generators
    memory_profiler.py       ← Resource usage monitoring
    format_validator.py      ← Output format compliance checking
  tests/
    debug_test_cases/        ← Problematic cases for debugging
      edge_cases.py          ← Empty files, identical files, etc.
      large_files.py         ← Memory and performance stress tests
      malformed_input.py     ← Binary files, encoding issues
  src/
    diff_tool/
      debug_hooks.py         ← Integration points for debugging tools
```

Debugging Infrastructure Starter Code

Complete Matrix Visualizer:

```python
"""
Matrix visualization and validation tools for LCS debugging.
"""
import sys
from typing import List, Optional, Tuple


class MatrixVisualizer:
    """Provides visualization and validation for LCS dynamic programming matrices."""


    def __init__(self, seq1: List[str], seq2: List[str], matrix: List[List[int]]):
        self.seq1 = seq1
        self.seq2 = seq2
        self.matrix = matrix


    def display_matrix(self) -> str:
        """Generate a formatted matrix display with sequence headers."""
        # Calculate column widths for proper alignment
        col_widths = [max(3, len(str(val))) for val in ['']+self.seq2]
        row_headers = ['']+self.seq1


        # Build header row
        header = "".join(f"{val:>{width}}" for val, width in zip([''] + self.seq2,
col_widths))
        lines = [header]


        # Build data rows
        for i, row_header in enumerate(row_headers):
            row_data = [row_header] + [str(self.matrix[i][j]) for j in
range(len(self.matrix[i]))]
```

```
            line = "".join(f"{val:>{width}}" for val, width in zip(row_data, col_widths))
            lines.append(line)

        return "\n".join(lines)

    def validate_matrix_properties(self) -> List[str]:
        """Validate that matrix satisfies LCS properties."""
        errors = []

        # Check dimensions
        expected_rows = len(self.seq1) + 1
        expected_cols = len(self.seq2) + 1
        if len(self.matrix) != expected_rows:
            errors.append(f"Wrong row count: got {len(self.matrix)}, expected
{expected_rows}")
        if any(len(row) != expected_cols for row in self.matrix):
            errors.append(f"Inconsistent column count: expected {expected_cols}")

        # Check boundary conditions (first row and column should be zeros)
        if any(self.matrix[0][j] != 0 for j in range(len(self.matrix[0]))):
            errors.append("First row should be all zeros")
        if any(self.matrix[i][0] != 0 for i in range(len(self.matrix))):
            errors.append("First column should be all zeros")

        # Check monotonicity (values never decrease moving right or down)
        for i in range(1, len(self.matrix)):
            for j in range(1, len(self.matrix[i])):
                current = self.matrix[i][j]
```

```
                left = self.matrix[i][j-1]

                top = self.matrix[i-1][j]



                if current < left:

                    errors.append(f"Value decreased moving right at ({i},{j}): {left} ->
{current}")

                if current < top:

                    errors.append(f"Value decreased moving down at ({i},{j}): {top} ->
{current}")



        return errors


class BacktrackTracer:

    """Traces backtracking path through LCS matrix for debugging."""



    def __init__(self, matrix: List[List[int]], seq1: List[str], seq2: List[str]):

        self.matrix = matrix

        self.seq1 = seq1

        self.seq2 = seq2



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

        """Perform backtracking with detailed logging of each decision."""

        lcs = []

        path = []



        i, j = len(self.seq1), len(self.seq2)



        while i > 0 and j > 0:

            current = self.matrix[i][j]
```

```
            if self.seq1[i-1] == self.seq2[j-1]:
                # Characters match - move diagonally
                lcs.append(self.seq1[i-1])
                path.append((i, j, f"MATCH: '{self.seq1[i-1]}' - diagonal to ({i-1},{j-1})"))
                i -= 1
                j -= 1
            elif self.matrix[i-1][j] > self.matrix[i][j-1]:
                # Deletion from seq1 - move up
                path.append((i, j, f"DELETE seq1[{i-1}]='{self.seq1[i-1]}' - up to ({i-1},
{j})"))
                i -= 1
            else:
                # Insertion into seq1 - move left
                path.append((i, j, f"INSERT seq2[{j-1}]='{self.seq2[j-1]}' - left to ({i},{j-
1})"))
                j -= 1

        # Handle remaining elements
        while i > 0:
            path.append((i, j, f"DELETE remaining seq1[{i-1}]='{self.seq1[i-1]}'"))
            i -= 1
        while j > 0:
            path.append((i, j, f"INSERT remaining seq2[{j-1}]='{self.seq2[j-1]}'"))
            j -= 1

        lcs.reverse()  # Backtracking builds LCS in reverse order
        return lcs, path
```

Complete Performance Monitor:

```python
"""
Performance and resource monitoring for diff algorithm debugging.
"""
import time
import psutil
import gc
from typing import Dict, Any, Optional
from dataclasses import dataclass

@dataclass
class PerformanceStats:
    """Statistics collected during algorithm execution."""
    start_time: float
    end_time: Optional[float]
    peak_memory_mb: float
    operations_count: int
    matrix_size: Tuple[int, int]
    algorithm_phase: str


    @property
    def elapsed_time(self) -> float:
        if self.end_time is None:
            return time.time() - self.start_time
        return self.end_time - self.start_time

class PerformanceMonitor:
    """Monitors resource usage during algorithm execution."""


    def __init__(self):
```

```
        self.stats: Dict[str, PerformanceStats] = {}
        self.current_phase: Optional[str] = None

    def start_phase(self, phase_name: str, matrix_size: Tuple[int, int] = (0, 0)):
        """Begin monitoring a specific algorithm phase."""
        if self.current_phase:
            self.end_phase()

        self.current_phase = phase_name
        gc.collect()  # Clean slate for memory measurement

        self.stats[phase_name] = PerformanceStats(
            start_time=time.time(),
            end_time=None,
            peak_memory_mb=self._get_memory_usage_mb(),
            operations_count=0,
            matrix_size=matrix_size,
            algorithm_phase=phase_name
        )

    def end_phase(self):
        """Complete monitoring of current phase."""
        if not self.current_phase:
            return

        stats = self.stats[self.current_phase]
        stats.end_time = time.time()
        stats.peak_memory_mb = max(stats.peak_memory_mb, self._get_memory_usage_mb())
```

```
        self.current_phase = None

    def record_operation(self, count: int = 1):
        """Record completion of algorithm operations (e.g., matrix cell computations)."""
        if self.current_phase:
            self.stats[self.current_phase].operations_count += count
            # Update peak memory periodically
            if self.stats[self.current_phase].operations_count % 1000 == 0:
                current_memory = self._get_memory_usage_mb()
                self.stats[self.current_phase].peak_memory_mb = max(
                    self.stats[self.current_phase].peak_memory_mb,
                    current_memory
                )

    def get_report(self) -> Dict[str, Any]:
        """Generate performance report for all monitored phases."""
        report = {}

        for phase_name, stats in self.stats.items():
            complexity_score = "N/A"
            if stats.matrix_size[0] > 0 and stats.matrix_size[1] > 0:
                expected_ops = stats.matrix_size[0] * stats.matrix_size[1]
                if expected_ops > 0:
                    complexity_score = f"{stats.operations_count / expected_ops:.2f}x
expected"

            report[phase_name] = {
                "elapsed_time": stats.elapsed_time,
```

```
            "peak_memory_mb": stats.peak_memory_mb,
            "operations": stats.operations_count,
            "matrix_size": stats.matrix_size,
            "complexity_score": complexity_score
        }

    return report

def _get_memory_usage_mb(self) -> float:
    """Get current process memory usage in MB."""
    process = psutil.Process()
    return process.memory_info().rss / 1024 / 1024
```

Core Debugging Integration Points

Debug-Enhanced LCS Engine:

PYTHON

```python
class DebugLCSEngine(LCSEngine):
    """LCS engine with debugging capabilities enabled."""

    def __init__(self, debug_enabled: bool = False):
        super().__init__()
        self.debug_enabled = debug_enabled
        self.visualizer: Optional[MatrixVisualizer] = None
        self.tracer: Optional[BacktrackTracer] = None
        self.monitor = PerformanceMonitor()

    def compute_lcs(self, sequence1: Sequence, sequence2: Sequence) -> CommonSubsequence:
        """Compute LCS with optional debugging instrumentation."""
        # TODO 1: Start performance monitoring for matrix construction phase
        # TODO 2: Build LCS matrix with debug logging if enabled
        # TODO 3: Create matrix visualizer if debug mode active
        # TODO 4: Validate matrix properties and report any violations
        # TODO 5: Start performance monitoring for backtracking phase
        # TODO 6: Perform backtracking with path tracing if enabled
        # TODO 7: Generate performance report and debug output
        # TODO 8: Return CommonSubsequence with debug metadata attached
        pass

    def _debug_matrix_construction(self, seq1: Sequence, seq2: Sequence) -> List[List[int]]:
        """Build matrix with detailed logging for debugging."""
        # TODO 1: Log matrix dimensions and expected memory usage
        # TODO 2: Initialize matrix with boundary condition logging
        # TODO 3: Fill matrix with cell-by-cell operation logging
        # TODO 4: Validate intermediate results every N operations
```

```
        # TODO 5: Check for memory usage spikes during construction
        pass


    def _validate_and_report(self):
        """Generate comprehensive debugging report."""
        # TODO 1: Run matrix property validation
        # TODO 2: Generate matrix visualization if requested
        # TODO 3: Create backtracking trace log
        # TODO 4: Compile performance statistics
        # TODO 5: Write debug report to configured output stream
        pass
```

Language-Specific Debugging Hints

Python Debugging Tools:

- Use `pdb.set_trace()` for interactive debugging at algorithm decision points
- Install `memory_profiler`: `pip install memory_profiler` for line-by-line memory analysis
- Use `sys.getsizeof()` to measure data structure memory usage
- Enable warnings: `python -W all diff_tool.py` to catch potential issues
- Use `tracemalloc` module for detailed memory allocation tracking

Matrix Indexing Safety:

- Always use `range(1, len(seq)+1)` for matrix filling loops to avoid off-by-one errors
- Add assertions: `assert 0 <= i < len(matrix)` before every matrix access
- Use `matrix[i][j] if i < len(matrix) and j < len(matrix[i]) else 0` for safe access

File Handling Edge Cases:

- Test with `os.devnull` as input to verify empty file handling
- Use `tempfile.NamedTemporaryFile()` for creating test files with specific encodings
- Test with files lacking trailing newlines: `echo -n "content" > test.txt`

Milestone Checkpoint Integration

After Milestone 1 (Line Tokenization): Expected behavior: `python debug/test_tokenization.py` should validate file reading with various encodings and line endings. Manual verification: create files with different encodings and verify line counts match `wc -l` output.

After Milestone 2 (LCS Algorithm):
Expected behavior: `python debug/matrix_validator.py` should confirm matrix properties hold for test cases. Debug output should show matrix visualization for small examples. Performance monitor should report O(mn) complexity scaling.

After Milestone 3 (Diff Generation): Expected behavior: `python debug/format_validator.py` should verify unified diff format compliance. Round-trip tests should successfully reconstruct target files from diff operations.

After Milestone 4 (CLI and Color Output): Expected behavior: All debugging tools should work through CLI interface. Color output should validate correctly in both TTY and pipe modes. Exit codes should match diff result status.

Future Extensions

> **Milestone(s):** All milestones — future extensions build upon the complete foundation from line tokenization (Milestone 1), LCS computation (Milestone 2), diff generation (Milestone 3), and CLI output (Milestone 4)

The current diff tool design provides a solid foundation that can accommodate numerous enhancements while maintaining architectural integrity. Like building a house with a strong foundation that can support additional floors, our component-based architecture with clear separation between `FileReader`, `LCSEngine`, `DiffGenerator`, and `OutputFormatter` allows for incremental improvements without requiring fundamental restructuring.

The extension strategy follows a principle of **progressive enhancement** — each improvement builds upon existing capabilities while preserving backward compatibility. This approach ensures that basic diff functionality remains stable while advanced features can be added, disabled, or modified independently. The modular design means that algorithm improvements can be implemented as alternative strategies within the `LCSEngine`, while output enhancements can be added as new formatters alongside the existing `OutputFormatter`.

Algorithm Improvements

The current LCS-based approach represents just one point in the rich landscape of diff algorithms. Like choosing between different route-finding algorithms for navigation — some optimize for shortest distance, others for fastest time, others for avoiding traffic — different diff algorithms optimize for different characteristics. Our architecture anticipates this diversity by encapsulating algorithm choice within the `LCSEngine` component.

Myers' Algorithm Implementation

Mental Model: The Shortest Edit Path
Think of text comparison as finding the shortest path through a maze where each cell represents a comparison between characters or lines. Myers' algorithm is like having a GPS that explores multiple paths simultaneously, always pursuing the most promising routes first. Instead of filling out the entire maze (like our current LCS matrix approach), Myers' algorithm uses an **edit graph** where diagonal moves represent matches and horizontal/vertical moves represent insertions/deletions.

The algorithm maintains a concept of **edit distance** — the minimum number of operations needed to transform one sequence into another. Myers' algorithm achieves $O(n+d^2)$ expected performance where d is the actual edit distance, making it significantly faster than O(mn) LCS when files are similar.

> **Decision: Myers' Algorithm Integration Strategy**
>
> - **Context**: Current O(mn) LCS algorithm becomes prohibitively slow for large files, especially when most content is similar
> - **Options Considered**: Replace LCS entirely, implement Myers' as alternative strategy, hybrid approach
> - **Decision**: Implement Myers' as selectable strategy within existing `LCSEngine` architecture
> - **Rationale**: Preserves backward compatibility while providing performance benefits; allows algorithm comparison and fallback behavior
> - **Consequences**: Enables handling larger files efficiently while maintaining simple LCS for educational purposes and edge cases

Algorithm Aspect	LCS Dynamic Programming	Myers' Algorithm	Implementation Impact
Time Complexity	O(mn) always	$O(n+d^2)$ expected	Myers' much faster for similar files
Space Complexity	O(mn) or O(min(m,n)) optimized	O(n+d)	Myers' uses less memory
Implementation Complexity	Straightforward DP table	Complex edit graph traversal	Myers' requires more sophisticated code
Educational Value	Excellent for learning DP	Advanced algorithm concepts	Keep both for different learning objectives
Worst-Case Behavior	Predictable O(mn)	Can degrade to O(mn) for very different files	Need fallback strategy

The `LCSEngine` would be enhanced with a strategy selection mechanism. The `choose_strategy(seq1_len, seq2_len)` method would analyze input characteristics and select the optimal algorithm. For files under 1000 lines or when educational mode is enabled, the system would use traditional LCS. For larger files, Myers' algorithm would be preferred.

```python
# Enhanced LCSEngine with algorithm selection

def choose_strategy(self, seq1_len: int, seq2_len: int) -> MemoryStrategy:

    estimated_lcs_memory = (seq1_len * seq2_len * 8) / (1024 * 1024)  # MB



    if self.educational_mode or seq1_len < 1000 or seq2_len < 1000:

        return MemoryStrategy(algorithm="lcs_matrix", max_memory_mb=estimated_lcs_memory,

                              estimated_time_factor=1.0)

    elif estimated_lcs_memory > self.memory_optimizer.max_memory_mb:

        return MemoryStrategy(algorithm="myers", max_memory_mb=estimated_lcs_memory * 0.1,

                              estimated_time_factor=0.3)

    else:

        return MemoryStrategy(algorithm="lcs_matrix", max_memory_mb=estimated_lcs_memory,

                              estimated_time_factor=1.0)
```

The Myers' implementation would require a new `EditGraph` data structure and `MyersEngine` class that implements the same interface as the current LCS approach but uses fundamentally different internal algorithms.

Word-Level and Character-Level Diffing

Mental Model: Zoom Levels in Image Editing

Current line-level diffing is like viewing an image at 100% zoom — you see the overall structure but miss fine details. Word-level diffing is like zooming to 200% to see individual pixels, while character-level diffing is like examining the image at 400% magnification. Each zoom level reveals different types of changes that matter for different use cases.

Word-level diffing becomes essential when comparing prose documents, configuration files, or code where line boundaries don't align with logical changes. A developer who renames a variable or fixes a typo shouldn't see the entire line marked as deleted and re-added — they should see the specific word that changed highlighted within the line context.

Decision: Hierarchical Diffing Architecture

- **Context**: Line-level diffing misses granular changes within lines, making output less readable for small modifications
- **Options Considered**: Replace line diffing with word diffing, implement separate word-diff tool, hierarchical approach
- **Decision**: Implement hierarchical diffing where line-level diff identifies changed lines, then word-level diff analyzes differences within those lines
- **Rationale**: Provides both structural overview and detailed changes; leverages existing line-level infrastructure
- **Consequences**: More complex output format but significantly improved usability for code review and document editing

The architecture would extend the `DiffGenerator` component with a `HierarchicalDiffGenerator` that performs multi-level analysis:

Diff Level	Input Unit	Algorithm	Use Case	Output Enhancement
Line	File lines	Current LCS/Myers'	Structural changes	Current unified diff format
Word	Line tokens split by whitespace/punctuation	LCS on word sequences	Variable renames, text editing	Highlighted words within unchanged lines
Character	Individual characters	Character-level LCS	Typo fixes, small edits	Precise change highlighting
Semantic	AST nodes (future)	Tree diff algorithms	Code refactoring	Semantic change descriptions

The word-level implementation would introduce a `TokenizerEngine` that splits lines into meaningful units:

```python
class TokenizerEngine:

    def tokenize_line(self, line: str, mode: str) -> List[Token]:

        """Split line into tokens based on mode (word, character, semantic)"""

        # TODO 1: Apply appropriate tokenization strategy

        # TODO 2: Preserve whitespace information for reconstruction

        # TODO 3: Handle punctuation and special characters appropriately

        # TODO 4: Return Token objects with position and type metadata
```

Each `Token` would maintain enough information to reconstruct the original line while enabling granular diff analysis. The `DiffGenerator` would first perform line-level diffing, then apply word-level analysis to lines marked as changed, producing nested `DiffLine` objects that contain both line-level and word-level change information.

Semantic Diffing for Code

Mental Model: Understanding vs. Memorizing
Traditional diff algorithms are like students who memorize text without understanding — they notice every character change but miss the logical meaning. Semantic diffing is like a teacher who understands the subject matter and can recognize when two different explanations convey the same concept. For code, this means understanding that `if (x == true)` and `if (x)` are semantically equivalent even though they're textually different.

Semantic diffing requires parsing code into Abstract Syntax Trees (AST) and comparing structural relationships rather than textual representation. This approach can identify meaningful changes like algorithm modifications while ignoring cosmetic changes like formatting, variable renames that don't affect logic, or comment additions.

The implementation would extend our architecture with a `SemanticDiffEngine` that operates on AST representations:

Code Change Type	Traditional Diff View	Semantic Diff View	Business Value
Variable Rename	Every line changed	"Renamed variable: oldName → newName"	Ignore cosmetic changes
Function Move	Large deletion + addition	"Moved function: Class1 → Class2"	Focus on structural changes
Comment Addition	Line additions throughout	No change reported	Hide documentation updates
Whitespace/Formatting	Extensive line changes	No change reported	Ignore style-only changes
Algorithm Change	Mixed line changes	"Modified sorting algorithm"	Highlight logic changes

The semantic engine would require language-specific parsers but could integrate with existing tools like Python's `ast` module, JavaScript's Babel parser, or tree-sitter for multi-language support.

Output Format Enhancements

The current unified diff format serves as a solid foundation, but different use cases demand different presentation approaches. Like a newspaper that might present the same information as a headline, detailed article, or infographic depending on the audience, our diff tool should support multiple output formats optimized for different consumption patterns.

Side-by-Side Display

Mental Model: Parallel Reading
Think of side-by-side diff display like reading parallel translations of a book — you can see both versions simultaneously and easily correlate changes across the two texts. This is particularly valuable for code review where understanding the context around changes is crucial, or for document editing where you need to see both the original intent and the revised version.

Side-by-side display requires fundamentally different layout calculations compared to unified diff format. Instead of interleaving changes in a single column, the formatter must align corresponding sections and handle cases where insertions and deletions don't match up cleanly.

> **Decision: Responsive Side-by-Side Layout**
>
> - **Context**: Unified diff format becomes difficult to read for large changes; reviewers need to see both versions simultaneously
> - **Options Considered**: Fixed two-column layout, responsive width adaptation, horizontal scrolling
> - **Decision**: Implement responsive layout that adapts to terminal width with graceful degradation
> - **Rationale**: Maximizes readability across different terminal sizes while providing side-by-side benefits
> - **Consequences**: More complex rendering logic but significantly improved user experience for code review

The implementation would extend `OutputFormatter` with a `SideBySideFormatter` class:

Layout Challenge	Solution Approach	Implementation Notes
Terminal Width Detection	Query terminal size, fall back to 80 columns	Use `shutil.get_terminal_size()` with fallback
Column Width Calculation	Dynamic split based on content + minimum readability	Reserve space for line numbers and separators
Line Alignment	Match corresponding lines, insert blanks for unmatched	Complex alignment algorithm for deletions vs insertions
Long Line Handling	Wrap lines within column boundaries	Preserve indentation and syntax highlighting
Color Synchronization	Coordinate color schemes across both columns	Ensure consistent highlighting between sides

```python
class SideBySideFormatter(OutputFormatter):
    def format_hunk_side_by_side(self, hunk: Hunk, column_width: int) -> List[str]:
        """Format hunk in side-by-side layout with proper alignment"""
        # TODO 1: Calculate optimal column widths based on terminal size
        # TODO 2: Align old and new lines, inserting blanks for unmatched content
        # TODO 3: Handle long lines by wrapping within column boundaries
        # TODO 4: Apply consistent color schemes to both columns
        # TODO 5: Add separator column with change indicators
```

The side-by-side formatter would also need to handle edge cases like very long lines, mixed content types, and terminal resizing during output generation.

HTML Output with Interactive Features

Mental Model: Document Publishing

Think of HTML diff output like converting a manuscript draft into a published article with interactive annotations. While command-line output serves developers working in terminals, HTML output serves broader audiences including project managers, documentation reviewers, and stakeholders who need to understand changes in a more accessible format.

HTML output enables rich interactivity that's impossible in terminal displays: collapsible sections, syntax highlighting, inline comments, change summaries, and navigation aids. This transforms the diff from a developer tool into a communication medium.

The architecture would add an `HTMLFormatter` that generates self-contained HTML documents:

HTML Feature	Technical Implementation	User Benefit
Syntax Highlighting	Integration with Pygments or highlight.js	Improved code readability
Collapsible Hunks	JavaScript accordions with CSS transitions	Focus on relevant changes
Change Statistics	Summary panels with charts	Quick overview of modification scope
Navigation Sidebar	Generated table of contents with anchor links	Easy movement through large diffs
Inline Annotations	Hover tooltips and expandable comments	Additional context without clutter
Export Options	Print-friendly CSS and PDF generation	Documentation and archival

```python
class HTMLFormatter(OutputFormatter):

    def generate_interactive_diff(self, file1: FileContent, file2: FileContent,

                                  hunks: List[Hunk]) -> str:

        """Generate complete HTML document with interactive diff features"""

        # TODO 1: Generate HTML structure with navigation and content areas

        # TODO 2: Apply syntax highlighting based on file extension detection

        # TODO 3: Create collapsible hunk sections with JavaScript controls

        # TODO 4: Add change statistics summary with visual indicators

        # TODO 5: Include CSS for responsive design and print compatibility

        # TODO 6: Embed JavaScript for interactive features without external dependencies
```

The HTML formatter would generate self-contained documents that work offline and can be shared easily. Integration with existing syntax highlighting libraries would provide language-aware formatting that surpasses terminal capabilities.

Integration with Version Control Systems

Mental Model: Native Git Citizen

Think of VCS integration like making our diff tool a native speaker of Git's language rather than a foreign translator. Instead of just comparing two arbitrary files, the tool would understand repository context, branch relationships, commit history, and merge conflicts. This transforms it from a generic comparison tool into a specialized Git companion.

Version control integration opens up powerful workflows: comparing working directory against specific commits, analyzing changes across branch merges, generating release notes from commit ranges, and providing enhanced conflict resolution during merges.

> **Decision: Git Protocol Integration**
>
> - **Context**: Developers primarily work within Git repositories where file comparison needs repository context
> - **Options Considered**: Shell wrapper scripts, native Git integration, separate VCS adapter layer
> - **Decision**: Implement VCS adapter pattern with Git as primary target, using libgit2 bindings
> - **Rationale**: Provides deep Git integration while maintaining extensibility for other VCS systems
> - **Consequences**: More complex dependency management but enables sophisticated repository-aware features

The implementation would add a `VCSAdapter` interface with Git-specific implementation:

VCS Feature	Implementation Approach	Diff Enhancement
Commit Range Comparison	Use libgit2 to traverse commit history	Compare any two points in project history
Branch Diff Analysis	Identify merge bases and divergence points	Show changes unique to each branch
Working Directory Integration	Monitor file status and staged changes	Compare working files against index or HEAD
Merge Conflict Resolution	Parse conflict markers and provide 3-way diff	Enhanced conflict visualization
Blame Integration	Correlate changes with commit authors and dates	Attribution information in diff output
Submodule Awareness	Handle submodule boundary detection	Proper handling of nested repositories

```python
class GitAdapter(VCSAdapter):

    def get_file_at_commit(self, filepath: str, commit_hash: str) -> FileContent:

        """Retrieve file content at specific commit for comparison"""

        # TODO 1: Use libgit2 to access repository object database

        # TODO 2: Resolve commit hash to tree object

        # TODO 3: Navigate tree to find file blob

        # TODO 4: Extract blob content with proper encoding detection

        # TODO 5: Create FileContent with commit metadata


    def compare_commit_range(self, start_commit: str, end_commit: str,

                             filepath: str) -> List[Hunk]:

        """Generate diff for file changes across commit range"""

        # TODO 1: Validate commit range and file existence

        # TODO 2: Retrieve file content at both commits

        # TODO 3: Apply standard diff pipeline to historical content

        # TODO 4: Annotate output with commit metadata
```

The VCS integration would also enable advanced features like **change velocity analysis** (tracking how frequently different parts of files change over time) and **collaboration patterns** (identifying areas where multiple developers

frequently make conflicting changes).

Advanced Output Customization

Mental Model: Custom Report Generation

Think of output customization like a newspaper editor who can present the same story as a front-page headline, detailed investigative piece, or statistical infographic depending on the audience and purpose. Different stakeholders need different levels of detail and different presentation formats from the same underlying diff analysis.

Advanced customization goes beyond simple formatting to include **content filtering**, **aggregation strategies**, and **presentation modes** tailored to specific workflows.

The architecture would support customizable output pipelines through a **template system** and **filter chain**:

Customization Type	Configuration Approach	Example Use Case
Content Filtering	YAML configuration files	Hide whitespace changes for code review
Aggregation Rules	Python expressions for grouping	Summarize changes by file type or author
Template Systems	Jinja2 templates for output format	Custom corporate report formats
Notification Integration	Webhook and email template support	Automated change notifications
Metrics Collection	Plugin system for change analysis	Code quality and complexity metrics

```python
class CustomizableOutputPipeline:

    def apply_filter_chain(self, hunks: List[Hunk],
                           filters: List[DiffFilter]) -> List[Hunk]:
        """Apply sequence of filters to diff output"""
        # TODO 1: Load filter configuration from YAML or Python modules
        # TODO 2: Apply each filter in sequence, allowing early termination
        # TODO 3: Support filter parameterization and conditional application
        # TODO 4: Maintain audit trail of applied filters


    def generate_from_template(self, template_name: str,
                               diff_data: DiffData) -> str:
        """Generate output using specified template"""
        # TODO 1: Load template from filesystem or embedded templates
        # TODO 2: Prepare template context with diff statistics and metadata
        # TODO 3: Apply template engine with safety restrictions
        # TODO 4: Post-process output for format-specific requirements
```

Implementation Strategy and Migration Path

The extension implementation follows a **progressive enhancement strategy** that maintains backward compatibility while enabling advanced features. Like adding floors to a building without disrupting the ground floor, each enhancement builds upon existing architecture without breaking current functionality.

Phase 1: Algorithm Enhancement (3-4 weeks)

1. Implement `StrategySelector` within existing `LCSEngine`
2. Add Myers' algorithm as alternative computation strategy
3. Create performance benchmarking suite to validate improvements
4. Implement algorithm fallback mechanisms for edge cases

Phase 2: Multi-Level Diffing (4-5 weeks)

1. Develop `TokenizerEngine` for word and character level analysis
2. Extend `DiffGenerator` with hierarchical diffing capabilities
3. Enhance `DiffLine` data model to support nested change information

4. Create unified API that maintains existing interface compatibility

Phase 3: Output Format Extensions (3-4 weeks)

1. Implement `SideBySideFormatter` with terminal width adaptation
2. Develop `HTMLFormatter` with syntax highlighting integration
3. Create template system for custom output formats
4. Add format auto-detection based on output destination

Phase 4: VCS Integration (5-6 weeks)

1. Design `VCSAdapter` interface with Git implementation
2. Integrate libgit2 bindings for repository access
3. Implement commit range comparison and working directory integration
4. Add merge conflict resolution enhancements

⚠️ Pitfall: Feature Creep Management
The abundance of possible enhancements can lead to **scope creep** where the simple diff tool becomes an overly complex Swiss Army knife. Each enhancement should be evaluated against core use cases and implemented as optional, configurable features that don't complicate the basic diff workflow. Maintain separate command-line flags and configuration options so users can opt into complexity rather than having it imposed.

⚠️ Pitfall: Performance Regression
Adding multiple algorithms and output formats can introduce performance overhead even when advanced features aren't used. Implement **lazy loading** and **strategy selection** so that basic diff operations maintain their current performance characteristics. The `choose_strategy` method should default to simple approaches for small inputs and only engage complex algorithms when they provide clear benefits.

⚠️ Pitfall: Configuration Complexity
Advanced customization can create a configuration nightmare where users need extensive setup before the tool becomes useful. Design configuration with **smart defaults** and **progressive disclosure** — the tool should work well without any configuration, provide simple options for common customizations, and only expose complex configuration for power users who explicitly seek advanced capabilities.

Implementation Guidance

The future extensions leverage modern Python ecosystem capabilities while maintaining the educational value of the core implementation. The extensions are designed as **optional enhancements** that students can explore after mastering the fundamental concepts.

Technology Recommendations

Extension Area	Simple Option	Advanced Option
HTML Generation	String templating with f-strings	Jinja2 template engine with Pygments syntax highlighting
VCS Integration	subprocess calls to git command	pygit2 (libgit2 bindings) for native Git access
Algorithm Selection	Simple if/else strategy selection	Strategy pattern with performance profiling
Configuration	Command-line arguments only	YAML configuration files with schema validation
Testing Extensions	Manual verification of output	Automated visual diff testing with image comparison

Recommended File Structure Enhancement

The extensions maintain the existing modular structure while adding specialized components:

```
diff-tool/
├── core/                    # Existing core components
│   ├── file_reader.py
│   ├── lcs_engine.py
│   ├── diff_generator.py
│   └── output_formatter.py
├── algorithms/              # Algorithm implementations
│   ├── __init__.py
│   ├── lcs_traditional.py   # Current implementation
│   ├── myers_algorithm.py   # Myers' diff algorithm
│   └── strategy_selector.py # Algorithm selection logic
├── formatters/              # Output format extensions
│   ├── __init__.py
│   ├── unified_formatter.py # Current implementation
│   ├── side_by_side.py      # Side-by-side display
│   ├── html_formatter.py    # HTML output with highlighting
│   └── template_engine.py   # Custom template support
├── vcs/                     # Version control integration
│   ├── __init__.py
│   ├── vcs_adapter.py       # Abstract VCS interface
│   ├── git_adapter.py       # Git-specific implementation
│   └── conflict_resolver.py # Enhanced merge conflict handling
├── tokenizers/              # Multi-level diffing support
│   ├── __init__.py
│   ├── line_tokenizer.py    # Current line-based approach
│   ├── word_tokenizer.py    # Word-level analysis
│   └── semantic_tokenizer.py # AST-based semantic diffing
├── config/                  # Configuration management
│   ├── __init__.py
│   ├── settings.py          # Configuration schema and defaults
│   └── templates/           # Built-in output templates
├── tests/
│   ├── extensions/          # Tests for extension components
│   ├── integration/         # End-to-end extension testing
│   └── performance/         # Algorithm performance benchmarks
└── examples/
    ├── custom_templates/    # Example template configurations
    └── vcs_integration/     # Git workflow examples
```

Algorithm Strategy Infrastructure

The strategy selection system provides a clean way to add new algorithms without breaking existing functionality:

PYTHON

```python
# algorithms/strategy_selector.py

from abc import ABC, abstractmethod

from typing import List, Tuple

from ..core.data_model import Sequence, CommonSubsequence, MemoryStrategy


class DiffAlgorithm(ABC):

    """Abstract interface for different diff algorithms"""



    @abstractmethod

    def compute_lcs(self, sequence1: Sequence, sequence2: Sequence) -> CommonSubsequence:

        """Compute longest common subsequence using this algorithm"""

        pass



    @abstractmethod

    def estimate_performance(self, seq1_len: int, seq2_len: int) -> Tuple[float, float]:

        """Return (time_factor, memory_mb) estimates for input size"""

        pass


class StrategySelector:

    """Selects optimal diff algorithm based on input characteristics"""



    def __init__(self):

        self.algorithms = {

            'lcs_matrix': TraditionalLCSAlgorithm(),

            'myers': MyersAlgorithm(),

            'hirschberg': HirschbergAlgorithm()

        }

        self.performance_history = {}  # Track actual performance for tuning
```

```python
    def choose_algorithm(self, seq1_len: int, seq2_len: int,
                         memory_limit_mb: float = 1024.0) -> str:
        """Select best algorithm for given constraints"""
        # TODO 1: Calculate performance estimates for each available algorithm
        # TODO 2: Filter algorithms that exceed memory constraints
        # TODO 3: Consider historical performance data for similar input sizes
        # TODO 4: Apply user preferences and educational mode settings
        # TODO 5: Return algorithm name with fallback to 'lcs_matrix'

        estimates = {}
        for name, algorithm in self.algorithms.items():
            time_factor, memory_mb = algorithm.estimate_performance(seq1_len, seq2_len)
            if memory_mb <= memory_limit_mb:
                estimates[name] = (time_factor, memory_mb)

        # Default to fastest algorithm that fits in memory
        if not estimates:
            return 'lcs_matrix'  # Fallback

        return min(estimates.keys(), key=lambda k: estimates[k][0])
```

HTML Output Infrastructure

The HTML formatter demonstrates how to create rich, interactive output while maintaining clean separation from core diff logic:

PYTHON

```python
# formatters/html_formatter.py

from typing import List, Dict, Any

import html

from ..core.data_model import FileContent, Hunk, DiffLine, LineType


class HTMLFormatter:

    """Generates interactive HTML diff displays"""


    def __init__(self, syntax_highlighting: bool = True,

                 interactive_features: bool = True):

        self.syntax_highlighting = syntax_highlighting

        self.interactive_features = interactive_features

        self.css_template = self._load_css_template()

        self.js_template = self._load_js_template()


    def generate_diff_document(self, file1: FileContent, file2: FileContent,

                               hunks: List[Hunk]) -> str:

        """Generate complete HTML document with embedded CSS and JavaScript"""

        # TODO 1: Generate HTML document structure with metadata

        # TODO 2: Embed CSS styles for diff formatting and responsive design

        # TODO 3: Create navigation sidebar with hunk links

        # TODO 4: Generate main diff content with syntax highlighting

        # TODO 5: Add interactive JavaScript for collapsible sections

        # TODO 6: Include print-friendly CSS media queries


        document_parts = [

            self._generate_html_header(file1.filepath, file2.filepath),

            self._generate_navigation_sidebar(hunks),
```

```
            self._generate_diff_content(file1, file2, hunks),
            self._generate_html_footer()
        ]

        return '\n'.join(document_parts)

    def _generate_diff_content(self, file1: FileContent, file2: FileContent,
                               hunks: List[Hunk]) -> str:
        """Generate main diff content area with syntax highlighting"""
        # TODO 1: Apply syntax highlighting based on file extension
        # TODO 2: Create hunk sections with collapsible controls
        # TODO 3: Format line numbers and change indicators
        # TODO 4: Add hover tooltips for additional context
        # TODO 5: Generate change statistics summary
        pass

    def _apply_syntax_highlighting(self, content: str, language: str) -> str:
        """Apply syntax highlighting using Pygments or similar"""
        # TODO 1: Detect language from file extension or content analysis
        # TODO 2: Use Pygments to generate highlighted HTML
        # TODO 3: Preserve diff markers and line structure
        # TODO 4: Handle edge cases like mixed languages or plain text
        pass
```

VCS Integration Foundation

The Git integration shows how to extend the diff tool with repository awareness:

PYTHON

```python
# vcs/git_adapter.py

import os

import subprocess

from typing import Optional, List, Tuple

from ..core.data_model import FileContent, DiffLine, Hunk

from .vcs_adapter import VCSAdapter


class GitAdapter(VCSAdapter):

    """Git-specific version control integration"""



    def __init__(self, repository_path: str):

        self.repo_path = repository_path

        self.git_cmd = ['git', '-C', repository_path]



    def get_file_at_commit(self, filepath: str, commit_hash: str) -> Optional[FileContent]:

        """Retrieve file content at specific commit"""

        # TODO 1: Validate that commit_hash exists in repository

        # TODO 2: Use 'git show commit:filepath' to get file content

        # TODO 3: Handle files that don't exist at specified commit

        # TODO 4: Detect encoding and normalize line endings

        # TODO 5: Create FileContent with commit metadata



        try:

            result = subprocess.run(

                self.git_cmd + ['show', f'{commit_hash}:{filepath}'],

                capture_output=True, text=True, check=True

            )

            return self._create_file_content_from_git(filepath, result.stdout, commit_hash)
```

```
        except subprocess.CalledProcessError:
            return None

    def compare_working_directory(self, filepath: str,
                                  against_commit: str = 'HEAD') -> List[Hunk]:
        """Compare working directory file against specified commit"""
        # TODO 1: Get current working directory file content
        # TODO 2: Get file content at specified commit
        # TODO 3: Apply standard diff pipeline to compare versions
        # TODO 4: Add Git-specific metadata to hunk headers
        pass

    def get_merge_conflicts(self, filepath: str) -> List[Tuple[int, int, str]]:
        """Parse merge conflict markers and return conflict regions"""
        # TODO 1: Read file and detect conflict markers (<<<<<<< ======= >>>>>>>)
        # TODO 2: Parse conflict regions into (start_line, end_line, conflict_type)
        # TODO 3: Extract 'ours', 'theirs', and 'base' content sections
        # TODO 4: Return structured conflict information for enhanced resolution UI
        pass
```

Milestone Checkpoints for Extensions

Each extension area provides clear validation steps:

Algorithm Enhancement Checkpoint:

- Run: `python -m diff_tool --algorithm=myers large_file1.txt large_file2.txt`
- Expected: Significantly faster execution on large files with similar content
- Validation: Performance improvement of 50%+ for files >10MB with <10% changes
- Debug check: Algorithm selection logging shows Myers' chosen for large inputs

HTML Output Checkpoint:

- Run: `python -m diff_tool --format=html file1.py file2.py > diff.html`

- Expected: Self-contained HTML file with syntax highlighting and interactive features
- Validation: Open diff.html in browser, verify collapsible hunks and color coding
- Debug check: HTML validates and includes embedded CSS/JavaScript

VCS Integration Checkpoint:

- Run: `python -m diff_tool --git-compare HEAD~1 HEAD src/main.py`
- Expected: Diff between current file and previous commit version
- Validation: Output includes commit metadata and matches `git diff HEAD~1 HEAD src/main.py`
- Debug check: Git repository detection and commit resolution working correctly

Common Extension Pitfalls

⚠️ Pitfall: Algorithm Selection Overhead
Adding multiple algorithms can introduce selection overhead that exceeds the benefits for small files. The strategy selector should cache performance characteristics and default to simple algorithms for inputs under 1000 lines. Measure actual selection time and ensure it's under 10ms for typical use cases.

⚠️ Pitfall: HTML Output Security
When generating HTML output, especially with user-provided content, ensure proper HTML escaping to prevent XSS vulnerabilities. Use `html.escape()` for all user content and avoid generating JavaScript from user input. The HTML should be safe to open in any browser without security warnings.

⚠️ Pitfall: Git Dependency Management
VCS integration can fail silently if Git isn't available or repository detection fails. Implement graceful degradation where VCS features are simply unavailable rather than causing tool failure. Provide clear error messages when Git operations fail and suggest fallback approaches.

⚠️ Pitfall: Configuration File Complexity
Advanced configuration options can overwhelm users and make the tool difficult to adopt. Design configuration with progressive disclosure: simple command-line flags for common options, optional configuration files for advanced users, and comprehensive defaults that work well without any configuration.

The extension architecture ensures that the diff tool can grow from a simple educational project into a sophisticated development tool while maintaining its core educational value and simplicity for basic use cases.

Glossary

> **Milestone(s):** All milestones — comprehensive terminology reference supporting line tokenization (Milestone 1), LCS computation (Milestone 2), diff generation (Milestone 3), and CLI output (Milestone 4)

The diff tool implementation involves specialized terminology from multiple domains including dynamic programming, text processing, file system operations, and command-line interface design. This glossary provides precise definitions for all technical terms, algorithms, and domain-specific vocabulary used throughout the design

document, organized to support both immediate reference during implementation and deeper understanding of the underlying concepts.

Core Algorithm Terms

Term	Definition	Context in Project
LCS	Longest Common Subsequence algorithm using dynamic programming to find the longest sequence of elements that appear in the same order in both input sequences, though not necessarily consecutively	Core algorithm in Milestone 2 for finding matching lines between files before generating diff output
dynamic programming	Algorithmic technique for solving optimization problems by building solutions from optimal subproblems, storing intermediate results to avoid recomputation	Foundation of LCS implementation in Milestone 2, where we build a matrix of optimal subsequence lengths
edit graph	Graph representation where moves represent edit operations - horizontal moves are deletions, vertical moves are insertions, diagonal moves are matches	Conceptual model underlying Myers' algorithm and visualizing the relationship between sequences
edit distance	Minimum number of operations needed to transform one sequence into another, typically counting insertions, deletions, and substitutions	Quantifies the difference between files and guides optimization decisions in diff generation
recurrence relation	Mathematical relationship defining how optimal solutions to subproblems combine to form solutions to larger problems	Defines how LCS matrix cells are computed based on neighboring cells and sequence element matches
optimal substructure	Property where optimal solutions to a problem contain optimal solutions to subproblems	Key property that makes dynamic programming applicable to LCS - optimal subsequences contain optimal shorter subsequences
backtracking	Process of reconstructing the optimal solution by tracing backwards through the dynamic programming matrix	Used in Milestone 2 to recover the actual LCS from the completed matrix of lengths
tie-breaking rules	Consistent handling of multiple valid LCS paths when backtracking through matrix cells with equal values	Ensures deterministic output when multiple equally valid longest common subsequences exist

Text Processing and File Handling Terms

Term	Definition	Context in Project
encoding detection	Process of determining the character encoding of a file by attempting to decode with common encodings like UTF-8 and Latin-1	Essential in Milestone 1 for reading files correctly before line comparison can begin
line normalization	Standardizing line endings, whitespace handling, and text structure while preserving meaningful content differences	Critical preprocessing step in Milestone 1 that affects all subsequent comparison accuracy
binary file detection	Identifying non-text files that contain binary data and cannot be meaningfully diffed as text	Prevents encoding errors and provides appropriate error messages when comparing unsupported file types
trailing newline	Final newline character at the end of a file, whose presence or absence can affect diff output	Common source of unexpected diff results that must be handled consistently across different editors and platforms
mixed line endings	File containing multiple line ending types (LF, CRLF, CR) which may indicate file corruption or cross-platform editing	Detected during line normalization in Milestone 1 and can indicate data integrity issues requiring user attention
UTF-8	Unicode encoding that can represent any character while maintaining ASCII compatibility	Primary encoding attempt in file reading, supporting international characters and modern text files
LATIN-1	Single-byte character encoding covering Western European languages	Fallback encoding when UTF-8 decoding fails, ensuring maximum compatibility with legacy text files
LF	Unix line ending character (0x0A) representing a single line feed	Standard line ending for Unix/Linux systems, requiring normalization for cross-platform compatibility
CRLF	Windows line ending sequence (0x0D 0x0A) combining carriage return and line feed	Standard line ending for Windows systems, must be normalized to prevent false differences
CR	Classic Mac line ending character (0x0D) representing a single carriage return	Legacy line ending from pre-OS X Mac systems, rarely encountered but must be handled for completeness

Diff Format and Output Terms

Term	Definition	Context in Project
unified diff	Standard diff output format using -/+ prefixes to mark deletions and additions, with context lines for readability	Target output format generated in Milestone 3, compatible with standard Unix diff tools and version control systems
context lines	Unchanged lines displayed around changes to provide readability and help locate modifications within the file	Configurable feature in Milestone 4 that affects hunk formation and overall diff readability
hunk	Group of consecutive changes with surrounding context lines, representing a logical block of modifications	Fundamental unit of diff output generated in Milestone 3, containing related changes grouped for human readability
edit script	Sequence of ADD, DELETE, and UNCHANGED operations that transform one file into another	Intermediate representation in Milestone 3 that bridges between LCS results and formatted hunk output
one-indexed	Line numbering convention starting at 1 rather than 0, used for human readability in diff output	Standard convention for diff format line numbers, requiring careful conversion from zero-indexed internal arrays
hunk header	@@ line range markers showing the position and size of changes in both files using format @@old_start,old_count +new_start,new_count@@	Essential component of unified diff format that allows diff consumers to locate and apply changes
file headers	--- and +++ lines at the beginning of diff output that identify the compared files	Standard unified diff format requirement that provides context about which files are being compared
edit operations	Classification of lines as ADD (inserted), DELETE (removed), or UNCHANGED (identical)	Core abstraction in Milestone 3 that represents the transformation between file versions

Command-Line Interface and Display Terms

Term	Definition	Context in Project
ANSI color codes	Terminal control sequences that format text with colors, bold, and other visual effects	Used in Milestone 4 to provide visual distinction between additions (green) and deletions (red)
TTY detection	Determining whether output is directed to a terminal or being piped to a file or another program	Critical for deciding when to apply ANSI color codes - colors should only appear in interactive terminal sessions
exit codes	Numeric values returned to the shell indicating program success (0) or failure (non-zero), with specific meanings	Standard convention where 0 indicates identical files and 1 indicates differences, used by scripts and automation
color mode	Configuration setting controlling when ANSI color formatting is applied to output	Allows users to force color on/off regardless of TTY detection, important for testing and special terminal scenarios
side-by-side display	Parallel presentation of both file versions showing changes in adjacent columns	Alternative output format that makes it easier to see before/after states of modified content

Data Structure and Architecture Terms

Term	Definition	Context in Project
pipeline pattern	Sequential processing architecture where data flows through distinct stages with well-defined interfaces	Architectural approach used for FileReader → LCSEngine → DiffGenerator → OutputFormatter flow
message-passing interface	Communication pattern where components interact through standardized method calls rather than shared state	Ensures clean separation between components and makes testing individual components easier
fail-fast strategy	Design principle of detecting and reporting errors immediately rather than allowing invalid state to propagate	Implemented throughout error handling to provide clear diagnostics and prevent cascading failures
contextual error propagation	Preserving error context and adding relevant information as errors bubble up through component layers	Ensures that error messages include enough context for users to understand and fix problems
progressive enhancement	Design approach that starts with basic functionality and adds advanced features without breaking core behavior	Guides the milestone progression from basic line comparison to advanced output formatting

Performance and Memory Management Terms

Term	Definition	Context in Project
memory optimization	Techniques to reduce space complexity for large inputs, such as using only two matrix rows instead of full matrix storage	Essential for handling large files in LCS computation without exhausting system memory
Hirschberg's algorithm	Space-efficient LCS algorithm using divide-and-conquer approach that achieves O(m+n) space complexity	Advanced optimization considered for very large file comparison when standard O(mn) approach fails
memory exhaustion	Condition where the algorithm attempts to allocate more memory than available, causing program failure	Primary risk when comparing large files using standard LCS matrix approach
resource limit enforcement	Preventing operations from exceeding system constraints through monitoring and early termination	Safety mechanism to prevent system instability when processing unexpectedly large inputs
graceful degradation	System behavior that maintains partial functionality when encountering resource constraints or errors	Design principle ensuring the tool provides useful output even when optimal algorithms cannot complete

Implementation and Testing Terms

Term	Definition	Context in Project
edge cases	Boundary conditions and unusual input scenarios that test the limits of algorithm correctness	Include empty files, identical files, files with no common lines, and files exceeding memory limits
off-by-one indexing	Common programming error involving incorrect array bounds or counting, especially critical in matrix operations	Frequent source of bugs in LCS matrix construction and line number calculations for diff output
hunk validation	Ensuring diff hunk consistency by verifying that line counts match actual content and ranges are correct	Quality assurance step that catches bugs in diff generation before output formatting
milestone checkpoints	Validation steps after completing each implementation stage to verify correct behavior before proceeding	Structured testing approach ensuring each component works correctly before building dependent components
property-based testing	Automated test generation that verifies algorithmic properties across many random inputs	Advanced testing strategy for validating LCS properties like optimality and consistency across different inputs
matrix inspection	Debugging technique involving visualization and validation of LCS dynamic programming matrices	Essential debugging tool for understanding why LCS algorithms produce unexpected results
round-trip testing	End-to-end verification that the complete pipeline preserves information and produces consistent results	Comprehensive testing approach ensuring the diff tool works correctly in real-world usage scenarios

Algorithm Complexity and Optimization Terms

Term	Definition	Context in Project
O(mn) time complexity	Algorithmic performance characteristic where execution time grows proportionally to the product of input sizes	Standard LCS algorithm performance - comparing files with m and n lines requires m×n operations
O(mn) space complexity	Memory usage that grows proportionally to the product of input sizes due to storing the full LCS matrix	Memory limitation that necessitates optimization techniques for large file comparison
matrix	Standard LCS algorithm using full O(mn) space to store complete dynamic programming table	Default implementation providing full algorithm transparency and debugging capability
two_row	Space optimization that uses only two matrix rows, reducing space complexity to O(min(m,n))	First-level optimization that significantly reduces memory usage while maintaining algorithm simplicity
hirschberg	Advanced space optimization using divide-and-conquer to achieve O(m+n) space complexity	Most advanced optimization for handling very large files when even two-row optimization insufficient

Error Classification and Handling Terms

Term	Definition	Context in Project
ERROR_CODE	Standardized error classification system enabling programmatic error handling and user guidance	Structured approach to error handling that provides specific error codes for different failure categories
FileSystemError	Error category for file access problems including missing files, permission issues, and I/O failures	Handles failures in Milestone 1 file reading operations with appropriate error messages and suggestions
EncodingError	Error category for character encoding detection and conversion failures	Addresses text decoding problems in Milestone 1 when files contain invalid character sequences
AlgorithmError	Error category for computational problems including matrix overflow and infinite loops	Covers failures in Milestone 2 LCS computation when algorithm encounters unexpected conditions
ResourceError	Error category for memory exhaustion, timeout, and system resource limitations	Prevents system instability when processing very large files exceeds available resources
OutputError	Error category for formatting and display problems including ANSI code issues and file write failures	Handles problems in Milestone 4 output generation and CLI interface operations

Development and Extension Terms

Term	Definition	Context in Project
scope creep	Uncontrolled expansion of project requirements that adds complexity without corresponding learning benefit	Avoided through clear non-goals definition and focus on core diff algorithm learning objectives
lazy loading	Design pattern that delays component initialization until actually needed, improving startup time and memory usage	Optimization strategy for loading heavyweight components like syntax highlighters only when required
smart defaults	Configuration that works well for most users without requiring manual setup or deep understanding	User experience principle ensuring the tool is immediately useful while supporting advanced customization
progressive disclosure	Interface design that reveals advanced features gradually rather than overwhelming users with options	Guides CLI design to present essential options prominently while making advanced features discoverable
algorithm fallback	Graceful degradation strategy where simpler algorithms are used when preferred approaches fail	Ensures reliability by providing backup computation strategies when resource-intensive algorithms cannot complete

Advanced Features and Algorithms Terms

Term	Definition	Context in Project
Myers algorithm	Efficient diff algorithm with $O(n+d^2)$ expected performance where d is the edit distance between sequences	Advanced algorithm suitable for future extension when basic LCS approach proves insufficient for large files
hierarchical diffing	Multi-level analysis that compares files at different granularities from lines to words to characters	Extension capability that can provide more precise diff information by analyzing changes at multiple levels
semantic diffing	Comparing code meaning rather than text representation by operating on abstract syntax trees	Advanced feature for code-aware diffing that understands programming language structure rather than treating code as plain text
VCS integration	Version control system awareness allowing the diff tool to work with Git, SVN, and other repositories	Extension path that enables the tool to compare file versions across commits and branches
strategy selection	Algorithm that chooses optimal diff approach based on input characteristics like file size and available memory	Intelligent system that automatically selects the best algorithm variant for given constraints

Data Structure and Type System Terms

Term	Definition	Context in Project
Sequence	Type alias for List[str] representing an ordered collection of lines for comparison	Fundamental type used throughout LCS computation and diff generation for consistent sequence handling
CommonSubsequence	Data structure containing the actual LCS elements along with their positions in both original sequences	Result type from LCS computation that provides both the subsequence and positional information for diff generation
EditDistance	Quantitative measure of file differences including counts of insertions, deletions, unchanged lines, and total operations	Provides statistical summary of changes that can guide performance optimization and user feedback
DiffLine	Representation of a single line in the diff output with content, type classification, and line numbers from both files	Core data structure in Milestone 3 that bridges between algorithm output and formatted display
LineType	Enumeration distinguishing between UNCHANGED, ADDED, and DELETED lines in the diff output	Type safety mechanism ensuring consistent handling of different line classifications throughout the pipeline
Hunk	Structured representation of a group of changes with context, including line ranges and formatting information	Primary output unit in Milestone 3 that packages related changes with sufficient context for human readability
FileContent	Complete representation of a file including filepath, lines, metadata, and encoding information	Comprehensive file abstraction from Milestone 1 that carries all necessary information through the diff pipeline

Performance Monitoring and Resource Management Terms

Term	Definition	Context in Project
MemoryStrategy	Configuration object specifying algorithm choice, memory limits, and performance trade-offs for LCS computation	Enables intelligent selection between different algorithm variants based on available resources and input size
ComputationStats	Performance metrics including execution time, memory usage, matrix dimensions, and algorithm selection	Provides detailed performance information for optimization and debugging of LCS computation
PerformanceMonitor	Component that tracks resource usage and execution statistics across different algorithm phases	Enables performance analysis and resource limit enforcement during expensive dynamic programming operations
ResourceMonitor	System that enforces memory and time limits to prevent resource exhaustion during large file processing	Safety mechanism that prevents system instability when processing unexpectedly large inputs
MAX_SAFE_LINES	Constant defining maximum line count to prevent integer overflow and memory exhaustion	Practical limit that guides when to apply memory optimizations or reject inputs as too large

Command-Line Interface and User Experience Terms

Term	Definition	Context in Project
DiffArguments	Container for parsed command-line arguments including file paths, context settings, and formatting options	Central configuration object in Milestone 4 that controls all aspects of diff tool behavior
ColorMode	Enumeration controlling when ANSI color codes are applied to output (always, never, auto based on TTY)	User preference system in Milestone 4 that balances visual enhancement with compatibility requirements
ColorFormatter	Utility class providing ANSI color code helpers for terminal text formatting	Encapsulates color handling logic in Milestone 4, supporting both colored and plain text output modes

Architecture and Design Pattern Terms

Term	Definition	Context in Project
PipelineContext	Execution context tracking current processing stage and progress for monitoring and error reporting	Provides visibility into diff tool execution progress and helps with debugging when operations fail
PipelineError	Exception type that preserves context information about where and why pipeline processing failed	Structured error handling that maintains enough information for meaningful error messages and debugging
DiffPipeline	Main coordinator component that orchestrates the flow from file reading through LCS computation to formatted output	Central control component that manages the interaction between FileReader, LCSEngine, DiffGenerator, and OutputFormatter

Debugging and Development Terms

Term	Definition	Context in Project
MatrixVisualizer	Debugging utility that generates formatted display of LCS matrices with sequence headers for inspection	Essential debugging tool for understanding LCS algorithm behavior and diagnosing incorrect results
BacktrackTracer	Debugging component that logs detailed information about backtracking decisions and path selection	Helps diagnose issues with LCS reconstruction by providing step-by-step trace of algorithm decisions
DebugLCSEngine	Enhanced LCS engine with additional debugging capabilities including matrix inspection and validation	Development tool that extends basic LCS engine with comprehensive debugging features for learning and troubleshooting
algorithm execution tracing	Step-by-step logging of algorithm decision-making process including matrix construction and backtracking	Debugging technique that provides detailed visibility into how the LCS algorithm processes input sequences
output format validation	Verification that generated diff output conforms to standard unified diff format requirements	Quality assurance process ensuring compatibility with existing tools and systems that consume diff output

Memory Management and Optimization Terms

Term	Definition	Context in Project
memory exhaustion	Condition where algorithm memory requirements exceed available system resources	Primary failure mode for large file comparison that necessitates algorithm selection and resource monitoring
space-time trade-off	Algorithm design decision balancing memory usage against computational complexity	Fundamental consideration in choosing between different LCS implementation strategies
memory-efficient backtracking	Techniques for reconstructing LCS without storing the complete matrix	Advanced optimization that enables LCS computation for very large files
incremental processing	Strategy of processing input in chunks rather than loading entire files into memory	Potential optimization for handling files larger than available system memory

Future Extension and Advanced Feature Terms

Term	Definition	Context in Project
Token	Granular unit of text (character, word, line) that serves as the basic comparison element	Abstraction that enables hierarchical diffing by supporting different levels of text granularity
TokenizerEngine	Component responsible for splitting text into meaningful comparison units based on selected granularity	Enables word-level and character-level diffing as extensions beyond basic line-based comparison
HierarchicalDiffGenerator	Extended diff generator that analyzes changes at multiple levels of granularity	Advanced feature that can provide more detailed change analysis by examining lines, words, and characters
SemanticDiffEngine	Diff engine that operates on abstract syntax trees rather than raw text	Code-aware diffing capability that understands programming language structure
SideBySideFormatter	Output formatter that presents changes in parallel columns rather than unified format	Alternative display format that makes before/after comparison more intuitive for human readers
HTMLFormatter	Output formatter that generates interactive web-based diff displays	Rich output format supporting folding, syntax highlighting, and interactive navigation
VCSAdapter	Abstract interface enabling integration with version control systems	Extension point for comparing file versions across commits and branches
StrategySelector	Component that analyzes input characteristics and selects optimal algorithm variant	Intelligent system that automatically chooses the best approach based on file size, available memory, and performance requirements

Quality Assurance and Validation Terms

Term	Definition	Context in Project
hunk consistency validation	Verification that hunk line counts accurately reflect the contained changes	Quality assurance check that prevents malformed diff output
LCS property verification	Testing that computed longest common subsequences satisfy mathematical correctness properties	Ensures algorithm implementation correctness through property-based validation
format compliance checking	Validation that generated output conforms to standard unified diff format specifications	Compatibility assurance ensuring output works with existing tools and systems
cross-platform compatibility	Ensuring consistent behavior across different operating systems and terminal environments	Critical requirement for a generally useful diff tool

This comprehensive glossary serves as both a reference during implementation and a learning resource for understanding the deeper concepts behind text comparison algorithms. Each term is defined with sufficient detail to understand its role in the overall system while maintaining connections to the specific milestones where these concepts become practically important.

The terminology spans from fundamental computer science concepts like dynamic programming and algorithmic complexity to practical implementation concerns like file encoding and terminal color handling. Understanding these terms and their relationships is essential for successfully implementing a robust diff tool that handles real-world text comparison scenarios effectively.

Implementation Guidance

A. Technology Recommendations Table:

Component	Simple Option	Advanced Option
CLI Parsing	`argparse` module with basic argument handling	`click` library with rich help formatting and command groups
File I/O	`open()` with try/except encoding detection	`pathlib` with `chardet` library for robust encoding detection
ANSI Colors	String constants with manual concatenation	`colorama` library for cross-platform color support
Performance Monitoring	Simple time/memory measurement	`memory_profiler` and `cProfile` for detailed analysis
Testing Framework	Built-in `unittest` module	`pytest` with fixtures and parameterized testing

B. Recommended Project Structure:

```
diff-tool/
├── src/
│   ├── __init__.py
│   ├── main.py                 # CLI entry point and argument parsing
│   ├── data_model.py           # Core types and data structures
│   ├── file_reader.py          # FileReader component
│   ├── lcs_engine.py           # LCS computation and optimization
│   ├── diff_generator.py       # Edit operations and hunk formation
│   ├── output_formatter.py     # Unified diff formatting and colors
│   ├── error_handling.py       # Error types and handling utilities
│   └── utils/
│       ├── __init__.py
│       ├── colors.py           # ANSI color utilities
│       ├── performance.py      # Performance monitoring
│       └── validation.py       # Data validation helpers
├── tests/
│   ├── __init__.py
│   ├── test_file_reader.py
│   ├── test_lcs_engine.py
│   ├── test_diff_generator.py
│   ├── test_output_formatter.py
│   └── fixtures/               # Test files with various encodings
│       ├── utf8_sample.txt
│       ├── latin1_sample.txt
│       └── binary_sample.bin
├── docs/
│   ├── design_document.md
│   └── api_reference.md
├── requirements.txt
├── setup.py
└── README.md
```

C. Infrastructure Starter Code:

File: `src/utils/colors.py` (Complete ANSI color utilities)

```python
import os

import sys

from enum import Enum


class ColorMode(Enum):

    """Controls when ANSI color codes are applied to output."""

    AUTO = "auto"      # Color when output is to TTY

    ALWAYS = "always"  # Force color regardless of output

    NEVER = "never"    # Never use color


# ANSI color constants

RED = "\033[31m"

GREEN = "\033[32m"

BOLD = "\033[1m"

DIM = "\033[2m"

RESET = "\033[0m"


class ColorFormatter:

    """ANSI color helper with TTY detection and mode control."""



    def __init__(self, color_mode: ColorMode = ColorMode.AUTO):

        self.color_mode = color_mode

        self._color_enabled = self._determine_color_enabled()



    def _determine_color_enabled(self) -> bool:

        """Determine if color should be enabled based on mode and TTY detection."""

        if self.color_mode == ColorMode.NEVER:

            return False

        elif self.color_mode == ColorMode.ALWAYS:
```

```
            return True
        else:  # AUTO mode
            return sys.stdout.isatty() and os.getenv('NO_COLOR') is None

    def red(self, text: str) -> str:
        """Format text with red ANSI codes if color enabled."""
        if self._color_enabled:
            return f"{RED}{text}{RESET}"
        return text

    def green(self, text: str) -> str:
        """Format text with green ANSI codes if color enabled."""
        if self._color_enabled:
            return f"{GREEN}{text}{RESET}"
        return text

    def bold(self, text: str) -> str:
        """Format text with bold ANSI codes if color enabled."""
        if self._color_enabled:
            return f"{BOLD}{text}{RESET}"
        return text

    def is_color_enabled(self) -> bool:
        """Check if color output is currently enabled."""
        return self._color_enabled
```

File: `src/utils/performance.py` (Complete performance monitoring)

PYTHON

```python
import time

import psutil

from dataclasses import dataclass

from typing import Dict, Optional, Tuple, Any


@dataclass

class PerformanceStats:

    """Performance statistics for a specific algorithm phase."""

    start_time: float

    end_time: Optional[float] = None

    peak_memory_mb: float = 0.0

    operations_count: int = 0

    matrix_size: Tuple[int, int] = (0, 0)

    algorithm_phase: str = ""


class PerformanceMonitor:

    """Monitors and reports performance statistics for diff operations."""



    def __init__(self):

        self.stats: Dict[str, PerformanceStats] = {}

        self.current_phase: Optional[str] = None

        self._process = psutil.Process()



    def start_phase(self, phase_name: str, matrix_size: Tuple[int, int] = (0, 0)) -> None:

        """Begin monitoring a specific algorithm phase."""

        self.current_phase = phase_name

        self.stats[phase_name] = PerformanceStats(

            start_time=time.time(),

            matrix_size=matrix_size,
```

```
            algorithm_phase=phase_name
        )

    def end_phase(self) -> None:
        """Complete monitoring of current phase."""
        if self.current_phase and self.current_phase in self.stats:
            stats = self.stats[self.current_phase]
            stats.end_time = time.time()
            stats.peak_memory_mb = self._get_memory_usage_mb()
            self.current_phase = None

    def record_operation(self, count: int = 1) -> None:
        """Record completion of algorithm operations."""
        if self.current_phase and self.current_phase in self.stats:
            self.stats[self.current_phase].operations_count += count

    def _get_memory_usage_mb(self) -> float:
        """Get current process memory usage in MB."""
        return self._process.memory_info().rss / (1024 * 1024)

    def get_report(self) -> Dict[str, Any]:
        """Generate comprehensive performance report."""
        report = {}
        for phase_name, stats in self.stats.items():
            duration = (stats.end_time - stats.start_time) if stats.end_time else 0
            report[phase_name] = {
                'duration_seconds': duration,
                'peak_memory_mb': stats.peak_memory_mb,
```

```
            'operations_count': stats.operations_count,
            'matrix_size': stats.matrix_size,
            'ops_per_second': stats.operations_count / duration if duration > 0 else 0
        }
    return report
```

File: `src/utils/validation.py` (Complete validation utilities)

PYTHON

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

def validate_hunk_consistency(hunk) -> bool:
    """Verify that hunk line counts match actual content."""
    # Count actual additions and deletions in hunk lines
    actual_deletions = sum(1 for line in hunk.lines if line.line_type.value == "DELETED")
    actual_additions = sum(1 for line in hunk.lines if line.line_type.value == "ADDED")
    actual_unchanged = sum(1 for line in hunk.lines if line.line_type.value == "UNCHANGED")

    # Calculate expected counts from hunk header
    expected_old_count = hunk.old_count
    expected_new_count = hunk.new_count

    # Validate that header counts match actual line content
    calculated_old = actual_deletions + actual_unchanged
    calculated_new = actual_additions + actual_unchanged

    return (calculated_old == expected_old_count and
            calculated_new == expected_new_count)

def normalize_line_endings(content: str) -> Tuple[str, str]:
    """Normalize line endings and detect original format."""
    # Detect original line ending format
    if '\r\n' in content:
        original_ending = 'CRLF'
    elif '\r' in content:
        original_ending = 'CR'
    elif '\n' in content:
```

```
        original_ending = 'LF'
    else:
        original_ending = 'NONE'

    # Normalize to LF
    normalized = content.replace('\r\n', '\n').replace('\r', '\n')
    return normalized, original_ending

def split_preserving_empty_lines(content: str) -> List[str]:
    """Split content into lines while preserving empty lines in sequence."""
    if not content:
        return []

    # Split on normalized line endings
    lines = content.split('\n')

    # Handle trailing newline - if content ends with newline,
    # split will create an empty string at the end that we should remove
    if content.endswith('\n') and lines and lines[-1] == '':
        lines.pop()

    return lines

def detect_binary_content(content: bytes, sample_size: int = 8192) -> bool:
    """Detect if file content appears to be binary rather than text."""
    # Check first sample_size bytes for null bytes or high percentage of non-printable chars
    sample = content[:sample_size]

    # Presence of null bytes strongly indicates binary content
```

```
    if b'\x00' in sample:
        return True

    # Count printable ASCII and common extended characters
    printable_count = sum(1 for byte in sample
                          if (32 <= byte <= 126) or byte in (9, 10, 13))

    # If less than 95% printable characters, likely binary
    if len(sample) > 0:
        printable_ratio = printable_count / len(sample)
        return printable_ratio < 0.95

    return False
```

D. Core Logic Skeleton Code:

File: `src/data_model.py` (Core types with method skeletons)

PYTHON

```python
from enum import Enum
from dataclasses import dataclass
from typing import List, Optional, Tuple


class LineType(Enum):
    """Classification of lines in diff output."""
    UNCHANGED = "UNCHANGED"
    ADDED = "ADDED"
    DELETED = "DELETED"


@dataclass
class DiffLine:
    """Represents a single line in diff output with metadata."""
    content: str
    line_type: LineType
    old_line_num: Optional[int]
    new_line_num: Optional[int]


    def format_unified_diff_line(self) -> str:
        """Format line with appropriate unified diff prefix."""
        # TODO 1: Get prefix character for line type (space, +, -)
        # TODO 2: Combine prefix with line content
        # TODO 3: Handle lines that don't end with newline
        pass


    def is_change(self) -> bool:
        """Returns True for ADD or DELETE lines."""
        # TODO 1: Check if line_type is ADDED or DELETED
        # TODO 2: Return False for UNCHANGED lines
```

```
        pass


@dataclass
class Hunk:
    """Group of consecutive changes with surrounding context."""
    old_start: int
    old_count: int
    new_start: int
    new_count: int
    lines: List[DiffLine]
    context_before: int
    context_after: int


    def format_hunk_header(self) -> str:
        """Generate @@ header line for unified diff format."""
        # TODO 1: Format old file range as start,count
        # TODO 2: Format new file range as +start,count
        # TODO 3: Combine into @@ -old_range +new_range @@
        # TODO 4: Handle special case where count is 1 (omit ,1)
        pass
```

E. Language-Specific Implementation Hints:

- **File Encoding**: Use `open(filepath, 'r', encoding='utf-8')` first, catch `UnicodeDecodeError` and retry with `encoding='latin-1'`
- **Memory Monitoring**: Use `psutil.Process().memory_info().rss` to get current memory usage in bytes
- **Cross-Platform Paths**: Use `pathlib.Path` instead of string concatenation for file path handling
- **ANSI Color Detection**: Check `sys.stdout.isatty()` and absence of `NO_COLOR` environment variable
- **Matrix Operations**: Use list comprehensions for matrix initialization: `[[0 for _ in range(n)] for _ in range(m)]`
- **Error Context**: Use `try/except` blocks that catch specific exceptions and re-raise with additional context
- **String Comparison**: Use `==` for exact line matching - Python handles Unicode comparison correctly

- **Performance Timing**: Use `time.perf_counter()` for high-resolution timing measurements

F. Milestone Checkpoint Verification:

Milestone	Command to Run	Expected Output	Manual Verification
1: Line Tokenization	`python -m pytest tests/test_file_reader.py -v`	All file reading tests pass	Create test files with different encodings, run `detect_file_encoding()`
2: LCS Algorithm	`python -m pytest tests/test_lcs_engine.py -v`	LCS computation tests pass with correct subsequences	Compare simple text files manually, verify LCS makes sense
3: Diff Generation	`python -m pytest tests/test_diff_generator.py -v`	Hunk generation tests pass with proper context	Generate diff for known files, check hunk boundaries and line numbers
4: CLI Integration	`python src/main.py file1.txt file2.txt`	Colored diff output in unified format	Test with `--no-color` flag, verify exit codes with `echo $?`

G. Common Implementation Patterns:

PYTHON

```python
# Error handling with context preservation
try:
    lines = read_file_lines(filepath)
except UnicodeDecodeError as e:
    raise EncodingError(
        f"Cannot decode file {filepath}",
        context={"filepath": filepath, "encoding_attempts": ["utf-8", "latin-1"]},
        suggestion="File may be binary or use unsupported encoding"
    )

# Resource monitoring pattern
monitor = ResourceMonitor(memory_limit_mb=512, time_limit_seconds=30)
monitor.start_monitoring()
try:
    result = expensive_operation()
    if error := monitor.check_limits():
        raise error
    return result
finally:
    stats = monitor.get_stats()


# Safe matrix construction with memory estimation
def estimate_matrix_memory(rows: int, cols: int) -> float:
    """Estimate LCS matrix memory requirements in MB."""
    # Each cell stores an integer (typically 8 bytes on 64-bit systems)
    bytes_needed = rows * cols * 8
    return bytes_needed / (1024 * 1024)
```