Build Your Own Spreadsheet: Design Document

Overview

This document outlines the architecture for an Excel-like spreadsheet application built with TypeScript/JavaScript in the browser. The core challenge is designing an efficient recalculation engine that maintains consistency across cell dependencies while providing responsive UI interactions. Key architectural decisions include a virtualized grid for rendering, a directed acyclic graph for dependency tracking, and a command pattern for undo/redo operations.

> 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.

> **Milestone(s):** All (provides foundational understanding for entire project)

Context and Problem Statement

Spreadsheet applications represent one of the most sophisticated and widely-used computational models in computing history. At its core, a spreadsheet is a **reactive dataflow system** disguised as a simple grid of cells. The fundamental challenge we must solve is: *How do we maintain mathematical consistency across thousands of interconnected formulas while providing real-time, interactive responsiveness to user changes?*

Unlike traditional programming where execution follows a predetermined sequence, spreadsheets must dynamically recompute values based on changing inputs, similar to how a circuit board propagates electrical signals. When a user types `=A1+B1` into cell C1, they're creating a **data dependency**—C1 now "listens" to changes in A1 and B1. If A1 later changes from 5 to 10, the system must detect this dependency, recalculate C1 (now 15), and then recalculate any cells that depend on C1, propagating the change through what could be a complex web of relationships.

Mental Model: The Spreadsheet as a Reactive Calculator

Imagine a spreadsheet as a massive **circuit board** with thousands of interconnected logic gates (cells). Each gate can either:

1. **Source cells:** Provide raw input values (like voltage sources)
2. **Logic cells:** Transform inputs through formulas (like AND/OR gates)
3. **Display cells:** Show the final output (like LEDs)

When you change one source cell, electricity flows through the circuit, updating all connected logic cells in the correct sequence until the display cells show the new stable state. Just as electricity follows physical laws, data in a spreadsheet must follow **dependency laws**—no cell can calculate its value until all its inputs are known.

A more accessible analogy is a **kitchen recipe network**:

- Each cell is a cooking station preparing one dish component
- Some stations start with raw ingredients (cell values like `5` or `"Sales"`)
- Others combine ingredients using recipes (formulas like `=SUM(A1:A10)`)
- Complex dishes depend on subcomponents (cell C10 depends on B5 which depends on A1)
- If you change the quality of tomatoes (A1), you must remake the sauce (B5) before you can remake the lasagna (C10)

This reactive model explains why spreadsheets feel magical—they give users a **declarative programming interface** where they specify *what* should be calculated, not *how* or *when* to calculate it. The engine handles the complex scheduling automatically.

Why Building a Spreadsheet is Hard

Building a production-grade spreadsheet engine involves solving four interconnected hard problems simultaneously:

1. **The Dependency Tracking Problem:** Formulas create directed relationships between cells. When A1 contains `=B1*2` and B1 contains `=C1+10`, changing C1 affects both B1 and A1. We must track these relationships to know what to recalculate, but naive tracking can miss indirect dependencies or create circular references.
2. **The Ordering Problem:** Cells must be recalculated in **topological order**—dependencies before dependents. If we calculate A1 (which needs B1) before B1 (which needs C1) before C1, we'll use stale values. The system must find the correct sequence through potentially thousands of nodes.
3. **The Performance Problem:** Recalculating the entire spreadsheet on every keystroke doesn't scale. Excel 95 introduced **natural order recalculation** that only updates affected cells, but even that can be slow with complex dependency graphs. We need incremental updates without sacrificing correctness.
4. **The Expression Problem:** Users can type arbitrary formulas like `=IF(A1>10, SUM(B1:B100), AVERAGE(C1:C100)*PI())`. The engine must parse these into executable computation trees, resolve cell references that might be relative (`A1`), absolute (`$A$1`), or ranged (`A1:B10`), and handle errors gracefully when formulas reference non-existent cells or create infinite loops.

The interdependence of these problems creates a **design tension**:

- Fast recalculation requires minimal dependency tracking, but correct tracking requires comprehensive graph analysis
- Simple parsing allows quick evaluation, but complex formulas require sophisticated AST construction
- Interactive editing demands immediate feedback, but large spreadsheets need batched updates

> **Key Insight:** The spreadsheet engine is essentially a **specialized database** with a computation layer. Cells are stored values, formulas are materialized views, and recalculation is view maintenance. This perspective explains why techniques from database systems (indexing, incremental view maintenance, transaction logs) apply directly to spreadsheet design.

Comparison of Existing Approaches

Different spreadsheet implementations have solved these problems with varying trade-offs. Understanding these approaches helps explain our architectural choices.

Approach	How It Works	Pros	Cons	Used In
Naive Full Recalc	Every cell is recalculated on every change, in arbitrary order	Simple to implement; Always correct for acyclic graphs	Exponential slowdown with grid size; Unusable for large sheets	Early spreadsheet programs (VisiCalc)
Natural Order Recalc	Cells recalculated in dependency order using topological sort	Only recalculates affected cells; Handles complex dependencies	Requires maintaining dependency graph; Circular reference detection needed	Excel 95, modern spreadsheets
Lazy Evaluation	Cells calculated only when needed for display or other formulas	Minimal computation; Fast for sparse sheets	"Uncalculated" state complexity; UI may show stale values	Some academic implementations
Incremental Recalc with Dirty Flags	Marks changed cells and dependents as "dirty," recalculates dirty cells in order	Very fast for small changes; Easy to implement	Complex to get right; May recalculate more than necessary	Google Sheets, our implementation
Push-Based Reactive	Changes propagate immediately through reactive programming primitives	Immediate updates; Elegant for simple cases	Hard to manage ordering; Circular reference prevention complex	Modern JavaScript frameworks (React+Signals)

Architecture Decision Records for Core Approach

Decision: Incremental Recalculation with Dependency Graph

- **Context**: We need a recalculation strategy that scales to 1000+ cells while maintaining interactive performance (<100ms per keystroke). Full recalc is too slow; lazy evaluation introduces UI complexity.
- **Options Considered**:
  1. **Full Recalculation Every Change**: Re-evaluate every formula on every change
  2. **Lazy/Demand-Driven Evaluation**: Only calculate cells when their value is needed (for display or by other formulas)
  3. **Incremental with Dirty Flag Propagation**: Track dependencies and only recalculate cells marked as "dirty"
- **Decision**: **Incremental with Dirty Flag Propagation** using a directed dependency graph with topological sort for ordering.
- **Rationale**:
  - **Performance**: Only recalculates the subset of cells affected by a change (typically <10% of all cells)
  - **Predictability**: Always maintains a fully calculated state (no "uncalculated" cells confusing users)
  - **Correctness**: Topological sort guarantees cells are calculated after their dependencies
  - **Debugability**: Dirty flags make it easy to trace what's being recalculated and why
- **Consequences**:
  - Must implement and maintain a dependency graph data structure
  - Need cycle detection to prevent infinite recalculation
  - Graph updates on every formula change add overhead
  - More complex than full recalc but scales better

Decision: Formula-as-String with AST Evaluation

- **Context**: Cells can contain formulas that reference other cells, use operators, and call functions. We need a way to parse and evaluate these formulas efficiently.
- **Options Considered**:
  1. **Store Raw Formula String**: Re-parse and evaluate on every recalculation
  2. **Compiled JavaScript Functions**: Convert formulas to JavaScript function objects
  3. **Abstract Syntax Tree (AST)**: Parse once to tree structure, evaluate tree multiple times
- **Decision**: **AST with dependency extraction during parsing**.
- **Rationale**:
  - **Parse Once, Evaluate Many**: Parsing is expensive; evaluation is cheap. AST allows parsing when formula changes, fast evaluation during recalc
  - **Dependency Extraction**: During parsing, we can identify all referenced cells and build the dependency graph
  - **Safety**: AST evaluation is safer than `eval()` or Function constructor (no arbitrary code execution)
  - **Debugging**: AST can be serialized for debugging formula logic
- **Consequences**:
  - AST storage overhead per formula cell
  - Must handle AST versioning when formulas change
  - More complex than string storage but better performance

Decision: Virtualized Grid Rendering

- **Context**: The spreadsheet grid can have 26+ columns × 100+ rows = 2600+ cells, but only ~20×10 = 200 fit on screen. Rendering all cells would crash the browser.
- **Options Considered**:
  1. **Render All Cells**: Create DOM elements for every cell in the grid
  2. **Canvas Rendering**: Draw cells on HTML Canvas element
  3. **Virtual Scrolling with DOM Recycling**: Only render visible cells, reuse DOM elements as user scrolls
- **Decision**: **Virtual Scrolling with DOM Recycling**.
- **Rationale**:
  - **Performance**: Creates only ~200 DOM elements instead of 2600+ (10× faster DOM creation, less memory)
  - **Native Interactions**: DOM elements support native text selection, input, CSS styling (unlike canvas)
  - **Smooth Scrolling**: Reusing DOM elements avoids layout thrashing during scroll
  - **Accessibility**: DOM elements work with screen readers and keyboard navigation
- **Consequences**:
  - Complex viewport-to-grid coordinate mapping
  - Must handle fast scrolling and jump-to-cell navigation
  - Selection highlighting needs special handling across viewport boundaries

Common Pitfalls in Spreadsheet Architecture

⚡ Pitfall: Naive String Formula Evaluation

- **Description**: Storing formulas as strings and using `eval()` or `Function()` constructor to evaluate them
- **Why It's Wrong**:

    - Security risk: Users could inject malicious JavaScript
    - Performance: Re-parsing on every recalculation is slow
    - No dependency extraction: Can't build a dependency graph
- **Fix**: Parse formula to AST once, extract dependencies during parse, evaluate AST

⚡ Pitfall: Recalculating Everything Every Time

- **Description**: When any cell changes, recalculating all formulas in the spreadsheet
- **Why It's Wrong**:
    - O(n) recalc time where n = number of formula cells
    - With 10,000 cells, typing becomes laggy
    - Wastes CPU on unaffected cells
- **Fix**: Build dependency graph, mark changed cells and dependents as "dirty," only recalculate dirty cells in topological order

⚡ Pitfall: Missing Circular Reference Detection

- **Description**: Allowing formulas like `A1 = B1+1` and `B1 = A1+1` without detection
- **Why It's Wrong**:
    - Infinite recalculation loop crashes browser tab
    - Maximum call stack exceeded errors
    - Can be subtle with longer dependency chains (A→B→C→A)
- **Fix**: When adding edges to dependency graph, check for cycles using depth-first search or topological sort failure detection

⚡ Pitfall: Fixed Grid Size in Memory

- **Description**: Allocating a 2D array for maximum possible grid size (e.g., 1000×1000 = 1M cells)
- **Why It's Wrong**:
    - Wastes memory on empty cells (most spreadsheets are sparse)
    - Hard to expand beyond initial allocation
    - Expensive to copy/expand array
- **Fix**: Use sparse data structure (Map of Maps or coordinate-based dictionary) that only stores occupied cells

⚡ Pitfall: Incorrect Reference Adjustment on Copy/Paste

- **Description**: When copying `=A1+B1` from C1 to D2, failing to adjust to `=B2+C2`
- **Why It's Wrong**:
    - Users expect relative references to adjust automatically
    - Absolute references (`=$A$1`) should NOT adjust
    - Mixed references (`=A$1`) should adjust partially
- **Fix**: Parse formula AST, identify references, adjust based on source→destination offset, respect `$` anchors

Implementation Guidance

Technology Recommendations Table:

Component	Simple Option	Advanced Option
Grid Rendering	DOM table with absolute positioning	Virtualized with DOM recycling using `transform: translate()`
Formula Parser	Recursive descent parser with regex tokenization	PEG.js or parser generator with grammar file
Dependency Graph	Adjacency list using `Map<string, Set<string>>`	Incremental topological sort using Kahn's algorithm with indegree tracking
Data Storage	Plain JavaScript object: `{[cellId]: value}`	Sparse column-store with compression
Undo/Redo	Command pattern with stack of cell changes	Operational transform with differential updates

Recommended File/Module Structure:

```
spreadsheet-engine/
├── src/
│   ├── core/
│   │   ├── index.ts              # Main export
│   │   ├── types.ts              # TypeScript interfaces and types
│   │   ├── spreadsheet.ts        # Main Spreadsheet class
│   │   └── cell.ts               # Cell data structure and metadata
│   ├── grid/
│   │   ├── virtual-grid.ts       # Virtualized grid rendering engine
│   │   ├── viewport-manager.ts   # Viewport-to-grid coordinate mapping
│   │   └── selection-manager.ts  # Cell selection and navigation
│   ├── formula/
│   │   ├── parser/
│   │   │   ├── tokenizer.ts      # Formula string → tokens
│   │   │   ├── parser.ts         # Tokens → AST
│   │   │   └── ast-nodes.ts      # AST node type definitions
│   │   ├── evaluator.ts          # AST evaluation with cell value resolution
│   │   └── functions.ts          # Built-in functions (SUM, AVG, etc.)
│   ├── graph/
│   │   ├── dependency-graph.ts   # Directed graph for cell dependencies
│   │   ├── topological-sort.ts   # Kahn's algorithm implementation
│   │   └── cycle-detector.ts     # DFS-based cycle detection
│   ├── features/
│   │   ├── undo-redo.ts          # Command pattern for undo/redo
│   │   ├── copy-paste.ts         # Copy/paste with reference adjustment
│   │   └── csv-export.ts         # CSV import/export utilities
│   └── utils/
│       ├── cell-utils.ts         # Cell address parsing (A1 → {row, col})
│       └── id-generator.ts       # Unique ID generation for operations
└── tests/                        # Test files mirroring src structure
```

Infrastructure Starter Code (Complete, Ready to Use):

Cell Address Utility (cell-utils.ts):

TYPESCRIPT

```typescript
/**
 * Convert column index (0-based) to Excel-style column letters (A, B, ..., Z, AA, AB, ...)
 */
export function columnToLetter(column: number): string {
  let temp = column;
  let letter = '';
  while (temp >= 0) {
    letter = String.fromCharCode((temp % 26) + 65) + letter;
    temp = Math.floor(temp / 26) - 1;
  }
  return letter;
}

/**
 * Convert Excel-style column letters to column index (0-based)
 */
export function letterToColumn(letters: string): number {
  let column = 0;
  for (let i = 0; i < letters.length; i++) {
    column = column * 26 + (letters.charCodeAt(i) - 64);
  }
  return column - 1;
}

/**
 * Parse cell address like "A1" or "$B$2" into structured format
 */
export interface ParsedCellReference {
  row: number;           // 0-based row index
  column: number;        // 0-based column index
  rowAbsolute: boolean;  // Whether row has $ prefix
  columnAbsolute: boolean; // Whether column has $ prefix
}
```

```
export function parseCellAddress(address: string): ParsedCellReference {
  // Matches optional $, column letters, optional $, row number
  const match = address.match(/^(\$?)([A-Z]+)(\$?)(\d+)$/i);
  if (!match) {
    throw new Error(`Invalid cell address: ${address}`);
  }

  const [, colPrefix, colLetters, rowPrefix, rowNumber] = match;
  return {
    row: parseInt(rowNumber, 10) - 1,
    column: letterToColumn(colLetters.toUpperCase()),
    rowAbsolute: rowPrefix === '$',
    columnAbsolute: colPrefix === '$'
  };
}

/**
 * Convert row and column to cell address string
 */
export function toCellAddress(row: number, column: number,
                              rowAbsolute: boolean = false,
                              columnAbsolute: boolean = false): string {
  const colStr = columnToLetter(column);
  const rowStr = (row + 1).toString();
  return `${columnAbsolute ? '$' : ''}${colStr}${rowAbsolute ? '$' : ''}${rowStr}`;
}

/**
 * Check if string is a valid cell address
 */
export function isValidCellAddress(address: string): boolean {
  return /^(\$?[A-Z]+\$?\d+)$/i.test(address);
}
```

Unique ID Generator (id-generator.ts):

```typescript
/**
 * Simple ID generator for operations and transactions
 */
export class IdGenerator {
  private counter = 0;
  private prefix: string;

  constructor(prefix: string = 'id_') {
    this.prefix = prefix;
  }

  next(): string {
    return `${this.prefix}${this.counter++}`;
  }

  reset(): void {
    this.counter = 0;
  }
}

// Singleton instance for spreadsheet operations
export const operationIdGenerator = new IdGenerator('op_');
export const transactionIdGenerator = new IdGenerator('tx_');
```

Language-Specific Hints for JavaScript/TypeScript:

1. **Use Maps for Sparse Data**: Instead of 2D arrays, use `Map<number, Map<number, Cell>>` where outer key is column, inner key is row
2. **Immutable Updates**: Use object/array spread or `structuredClone()` for undo/redo state snapshots
3. **Debounce User Input**: Use `lodash.debounce` or custom debounce for recalc triggers during rapid typing
4. **Web Workers for Heavy Computation**: Move formula evaluation or CSV processing to Web Workers to avoid UI blocking
5. **Custom Events for Communication**: Use `CustomEvent` for cross-component communication (grid ↔ formula engine ↔ graph)

Milestone Checkpoint for Context Understanding: Before starting implementation, verify you can answer these questions:

1. **Circuit Analogy**: If cell D4 contains `=A1+B2*C3`, draw the dependency circuit with arrows showing data flow

2. **Topological Sort**: Given cells A1=5, B1=`=A1+1`, C1=`=A1+B1`, D1=`=B1*2`, list the recalculation order if A1 changes to 10
3. **Performance Analysis**: Why does recalculating only dirty cells (incremental) scale better than full recalculation for a 1000×1000 grid?
4. **Error Scenario**: What happens with formulas `A1: =B1+1`, `B1: =C1*2`, `C1: =A1/3`? How should the system handle this?

Debugging Tips for Early Development:

Symptom	Likely Cause	How to Diagnose	Fix
Typing feels laggy	Full recalc instead of incremental	Console.log how many cells recalc on each keystroke	Implement dependency graph + dirty flag propagation
Wrong calculation results	Incorrect evaluation order	Log evaluation order and compare to dependency graph	Implement topological sort (Kahn's algorithm)
Browser freezes/crashes	Circular reference infinite loop	Add cycle detection when building dependency graph	Check for cycles when adding edges, show error to user
Formulas don't update when referenced cells change	Missing dependency tracking	Check if formula AST extracts and registers dependencies	Ensure parser collects cell references and graph updates on formula change

---

> **Milestone(s):** All (provides foundational understanding for entire project)

Goals and Non-Goals

Before diving into the technical architecture, it's essential to define the precise scope of the spreadsheet application. Building a feature-complete spreadsheet like Excel or Google Sheets is a monumental, multi-year effort. This project aims for an **educational implementation** that captures the **essential reactive dataflow** at the heart of all spreadsheets while intentionally excluding features that would distract from that core learning objective. Think of this as building the **engine and chassis of a car**—we focus on the propulsion system (formula recalculation) and basic steering (grid interaction), not the luxury interior (pivot tables, 3D charts, or macro scripting).

This section establishes a clear contract: what the system **must** achieve to be considered a functional spreadsheet engine, and what is explicitly **out of scope** to keep the project focused and achievable.

Must Have Features (Core Requirements)

The following features constitute the **minimum viable product (MVP)**. They are the non-negotiable capabilities that, when combined, deliver the fundamental spreadsheet experience. Each corresponds directly to one or more project milestones.

Milestone	Feature Category	Specific Requirement	Success Metric (How to verify it works)
1: Grid & Cell Rendering	**Interactive Grid**	Render a scrollable grid of at least `MAX_COLUMNS` (26) columns and `MAX_ROWS` (1000) rows.	Visual display of a labeled grid (A-Z headers, 1-1000 row numbers) that can be scrolled.
	Viewport Virtualization	Only render `VIEWPORT_ROWS` (20) x `VIEWPORT_COLUMNS` (10) cells in the DOM at any time, recycling nodes during scroll.	Smooth scrolling performance with no lag, observed via browser DevTools showing constant DOM node count.
	Cell Selection & Editing	Double-clicking a cell activates an inline text input for editing. A single selected cell is visually highlighted.	User can click, see a highlight border, double-click to edit, and press Enter to commit.
2: Formula Parser	**Formula Detection**	Any cell input starting with `=` is treated as a formula and parsed.	Typing `=5+3` into a cell and pressing Enter displays `8`.
	Basic Arithmetic	Support operators `+`, `-`, `*`, `/` with correct mathematical precedence (PEMDAS).	Formula `=2+3*4` evaluates to `14`, not `20`.
	Cell Reference Resolution	Formulas can reference other cells by address (e.g., `=A1+B2`).	If `A1` contains `5` and `B2` contains `3`, `=A1+B2` displays `8`.
	Built-in Functions	Implement `SUM(range)` that sums all numeric values in a contiguous rectangular range (e.g., `A1:A10`).	`=SUM(A1:A5)` correctly sums the values in cells A1 through A5.
3: Dependency Graph & Recalculation	**Reactive Updates**	Changing a cell's value automatically triggers recalculation of all cells that depend on it, directly or transitively.	If `B1` contains `=A1*2`, changing `A1` from `5` to `10` immediately updates `B1` to `20`.
	Correct Evaluation Order	The system uses a **topological sort** on the **dependency graph** to ensure cells are recalculated after their dependencies.	A chain `A1=1`, `B1=A1+1`, `C1=B1+1` recalculates in order A1→B1→C1 when A1 changes.
	Circular Reference Detection	The system detects dependency cycles (e.g., `A1 = B1+1` and `B1 = A1+1`) and shows an error without hanging.	Entering a circular reference displays a clear error (e.g., `#CIRCULAR!`) in the affected cells.
4: Advanced Features	**Copy/Paste with Reference Adjustment**	Copying a formula with relative references (e.g., `=A1+1`) and pasting it to the right adjusts the reference (`=B1+1`).	Copy cell C1 (`=A1+1`) to D1; D1's formula becomes `=B1+1`.
	Undo/Redo	A single level of undo (revert last change) and redo.	Change a cell's value, press Ctrl+Z, and the value reverts; Ctrl+Y re-applies it.
	CSV Export	Export the current grid data (values only, not formulas) to a downloadable `.csv` file.	Click "Export CSV" downloads a file that opens correctly in Excel or Numbers.
	Basic Number Formatting	Cells can be formatted as currency (`$1.23`), percentage (`12.3%`), or fixed decimals (`1.230`).	Applying "Currency" format to a cell with value `1.5` displays `$1.50`.

> **Design Insight:** The core "magic" of a spreadsheet is the **reactive dataflow** established by Milestones 2 and 3. Milestone 1 provides the necessary UI, and Milestone 4 adds polish and practical utility. If you were to ship only Milestones 1-3, you would have a functioning computational engine; Milestone 4 makes it feel like a real application.

To ensure these features are built on a solid foundation, the system must adhere to several **architectural invariants**:

1. **Single Source of Truth:** The `Spreadsheet` data model (containing the `grid` and `dependencyGraph`) is the sole authority for cell values and formulas. The UI is a derived, read-optimized view of this model.
2. **Deterministic Evaluation:** Given the same set of cell formulas and input values, the spreadsheet always produces the same output values. There is no randomness or time-dependent behavior in calculation.
3. **No Silent Data Loss:** User actions (edit, paste, undo) are either fully applied and reflected in the model and UI, or they fail with a clear error message. The application never enters a state where the UI shows one value but the model holds another.
4. **Bounded Resource Usage:** The grid is limited to `MAX_ROWS` and `MAX_COLUMNS`. The dependency graph cannot cause unbounded recursion or infinite loops (enforced by cycle detection).

Decision: Include Core Advanced Features in MVP

- **Context:** After implementing the reactive engine (Milestones 1-3), the application is functional but cumbersome for practical use. Basic productivity features like undo and copy/paste are expected in any interactive tool.
- **Options Considered:**
  1. **MVP = Milestones 1-3 only:** Deliver only the computational engine with minimal UI.
  2. **MVP = Milestones 1-3 + Select Advanced Features:** Include a curated set of advanced features essential for a usable demo.
  3. **MVP = All Proposed Milestones:** Implement everything listed in the project outline.
- **Decision:** Option 2 – Include a curated set from Milestone 4 (Copy/Paste with reference adjustment, single-level Undo/Redo, CSV Export, Basic Number Formatting) as part of the core requirements.
- **Rationale:** These four features have high utility-to-implementation-cost ratios. Copy/paste and undo are fundamental to user experience. CSV export provides tangible output. Basic formatting makes results readable. Together, they transform an academic engine into a demonstratable application without requiring excessive additional complexity (like multi-sheet support or full function libraries).
- **Consequences:**
  - **Enables:** A more polished, user-friendly demo that feels like a real product.
  - **Trade-offs:** Adds approximately 25% more implementation work beyond the core engine. Requires careful state management for undo/redo.

Option	Pros	Cons	Chosen?
MVP = Engine Only	Faster to build, focuses purely on the core algorithmic challenge.	Result feels incomplete and is frustrating to use interactively.	No
MVP + Curated Advanced Features	Strikes a balance between depth of learning and polish. Features like undo teach important patterns (Command).	Increases scope and requires implementing non-core features like file export.	**Yes**
MVP + All Advanced Features	Most impressive final product, covers a wide range of CS concepts.	Risk of project never finishing due to scope creep. Some features (e.g., full formatting) are very complex.	No

Explicitly Out of Scope

The following features are **intentionally excluded** from the project scope. This is not an exhaustive list of everything a commercial spreadsheet can do, but a deliberate boundary to keep the project focused on its educational goals around reactive dataflow and dependency management.

Feature Category	Specific Examples	Reason for Exclusion
Grid & UI Extensions	Multiple sheets/workbooks, freezing panes, row/column hiding, cell merging, rich text/styling within a cell, cell comments/notes.	These are UI/UX enhancements that do not directly contribute to learning the core formula evaluation and dependency graph algorithms. They add significant DOM and state management complexity.
Formula Language Extensions	String manipulation functions (`LEFT` , `CONCAT`), logical functions (`IF` , `AND`), lookup functions (`VLOOKUP`), date/time arithmetic, array formulas, user-defined functions (macros).	The parser and evaluator are designed to demonstrate the concept, not to be a full-featured language runtime. Adding string/logic/date types introduces significant type coercion and error handling complexity.
Advanced Recalculation	Intelligent/minimal recalculation (only recalc "dirty" cells), lazy evaluation, multi-threaded calculation, dependency analysis across sheets, iterative calculation for circular references (with limits).	The basic topological sort of the entire dependency subgraph on every change is sufficient to demonstrate the principle. Advanced recalculation is an optimization problem that distracts from the core graph algorithm.
Data Management	Database connections, pivot tables, charts/graphs, data validation rules, conditional formatting based on other cell values, "what-if" analysis tools.	These are application-level features built *on top* of a stable spreadsheet engine. They represent separate, large domains of functionality (graphics, database connectors).
Collaboration & Cloud	Real-time collaborative editing, change history/versioning, cloud auto-save, conflict resolution.	Introduces distributed systems challenges (networking, consensus, operational transformation) that are orthogonal to the single-user, in-memory engine focus.
Performance Optimizations	Incremental graph updates (adding/removing edges without full rebuild), memoization of cell values, persistent caching of evaluated results to disk, Just-In-Time (JIT) compilation of formulas.	While important for production systems, these optimizations obscure the fundamental, easier-to-understand algorithms we want to highlight.
Platform & Deployment	Mobile-responsive design, offline-first operation, installation as a Progressive Web App (PWA), cross-platform desktop wrapper (Electron).	These concern the delivery and packaging of the application, not the core spreadsheet logic.

> **Key Principle:** The primary learning objective is to understand and implement the **reactive dataflow system** where a change in one cell propagates through a **dependency graph** to update dependent cells in the correct order. Any feature that does not directly reinforce this objective is a candidate for exclusion.

Decision: Limit Formula Language to Arithmetic and SUM

- **Context:** The formula parser and evaluator must be powerful enough to demonstrate cell references and dependencies but simple enough to build in a limited time. A full expression language with many data types and functions is a project in itself.
- **Options Considered:**
  1. **Arithmetic Only:** Support only numbers, cell references, and `+` , `-` , `*` , `/` .
  2. **Arithmetic + Essential Functions:** Add a small set of numeric aggregate functions (`SUM` , `AVG` , `MIN` , `MAX` , `COUNT`).
  3. **Extended Language:** Add string, logical, and date functions, and support for parenthesized expressions.

- **Decision:** Option 2 – Support arithmetic and the `SUM` function as a representative built-in.
- **Rationale:** Arithmetic alone is sufficient for dependencies, but `SUM` introduces the critical concept of a **range reference** (`A1:A10`), which significantly affects dependency graph construction (one cell depends on *many* source cells). Implementing `SUM` provides the pattern for adding other functions later without the complexity of multiple data types.
- **Consequences:**
  - **Enables:** Demonstration of range dependencies and a template for future function addition.
  - **Trade-offs:** Requires parsing a new token type (the colon `:`) and handling a list of cells in the evaluator. More complex than arithmetic-only, but manageable.

Option	Pros	Cons	Chosen?
Arithmetic Only	Simplest parser and evaluator. Very fast to implement.	Cannot demonstrate range dependencies, feels extremely limited.	No
Arithmetic + SUM	Introduces range concept without type complexity. `SUM` is the universal "hello world" of spreadsheet functions.	Parser must handle the colon `:` operator and range validation.	**Yes**
Extended Language	More realistic and powerful. Teaches about type systems and function overloading.	Explodes scope: requires string/boolean types, error semantics, and many more parsing rules.	No

Common Pitfalls in Scoping

- ⚠️ **Pitfall: Creeping Feature Addition (Scope Creep)**
  - **Description:** While implementing the grid, you think, "It would be nice if users could resize columns," and you add that feature. Then you add row sorting, then cell coloring, and soon the core recalculation engine is still unfinished.
  - **Why it's wrong:** The educational goal is the dependency graph and recalculation engine. Every hour spent on a UI feature is an hour not spent on the primary learning objective. The project becomes a "pretty grid" instead of a "smart calculator."
  - **How to avoid:** Adhere strictly to the Must-Have Features table. Implement those completely *first*. Only after all core requirements are met and working should you consider adding any out-of-scope feature, and even then, treat it as a separate "bonus" phase.

Implementation Guidance

While the Goals and Non-Goals section is primarily about planning, establishing a clear project structure from the start is crucial for managing the defined scope.

Recommended File/Module Structure

Organize your codebase to mirror the architectural separation of concerns. This makes it easier to focus on one milestone at a time and prevents logic from becoming entangled.

```
spreadsheet-engine/
├── public/
│   └── index.html              # Single HTML page, includes main.js
├── src/
│   ├── index.js                # Application entry point, initializes everything
│   ├── model/                  # BUSINESS LOGIC LAYER (Milestones 2, 3, 4 logic)
│   │   ├── cell.js             # Cell data structure and methods
│   │   ├── formula-parser/     # Milestone 2
│   │   │   ├── tokenizer.js
│   │   │   ├── parser.js
│   │   │   ├── ast.js
│   │   │   └── evaluator.js    # Includes reference resolution & built-in functions
│   │   ├── dependency-graph/   # Milestone 3
│   │   │   ├── graph.js        # DependencyGraph class
│   │   │   └── topological-sort.js
│   │   └── advanced-features/ # Milestone 4
│   │       ├── command.js      # Command pattern for undo/redo
│   │       ├── clipboard.js    # Copy/paste with reference adjustment
│   │       └── formatter.js    # Number formatting
│   ├── view/                   # PRESENTATION LAYER (Milestone 1)
│   │   ├── grid.js             # Virtualized grid component, handles rendering & UI events
│   │   ├── viewport.js         # Manages the visible cell range
│   │   └── selection.js        # Tracks and highlights the selected cell
│   └── state/                  # STORAGE/STATE LAYER
│       ├── spreadsheet.js      # Main Spreadsheet class (composes Model, Graph, State)
│       └── history.js          # Undo/Redo stack (wraps Command pattern)
└── package.json
```

Infrastructure Starter Code

The following are helper modules that are not the core learning focus but are necessary prerequisites. You can copy these directly to establish a foundation.

1. **Cell Address Utilities (`src/model/cell-address.js`):** Provides the basic functions for converting between cell reference strings (A1) and row/column indices, as specified in the naming conventions.

JAVASCRIPT

```javascript
// src/model/cell-address.js

/**
 * Parse an Excel-style cell address into its components.
 * Supports relative (A1), absolute ($A$1), and mixed ($A1, A$1) references.
 * @param {string} address - The cell address string (e.g., "A1", "$B$2", "C$3")
 * @returns {ParsedCellReference} An object with row, column, and absolute flags.
 * @throws {Error} If the address format is invalid.
 */
export function parseCellAddress(address) {
    // Matches optional $, column letters, optional $, row numbers.
    const regex = /^(\$?)([A-Z]+)(\$?)(\d+)$/;
    const match = address.toUpperCase().match(regex);

    if (!match) {
        throw new Error(`Invalid cell address: "${address}"`);
    }

    const [, colAbs, colLetters, rowAbs, rowNum] = match;
    const column = letterToColumn(colLetters);
    const row = parseInt(rowNum, 10) - 1; // Convert to 0-based index

    if (row < 0 || row >= MAX_ROWS || column < 0 || column >= MAX_COLUMNS) {
        throw new Error(`Cell address out of bounds: "${address}"`);
    }

    return {
        row,
        column,
        rowAbsolute: rowAbs === '$',
        columnAbsolute: colAbs === '$'
    };
}

/**
```

```
 * Convert row and column indices back to a standard address string.
 * @param {number} row - 0-based row index.
 * @param {number} column - 0-based column index.
 * @param {boolean} rowAbsolute - If true, row part is absolute (e.g., $1).
 * @param {boolean} columnAbsolute - If true, column part is absolute (e.g., $A).
 * @returns {string} The cell address string.
 */
export function toCellAddress(row, column, rowAbsolute = false, columnAbsolute = false) {
    if (row < 0 || row >= MAX_ROWS || column < 0 || column >= MAX_COLUMNS) {
        throw new Error(`Indices out of bounds: row ${row}, column ${column}`);
    }
    const colPart = columnToLetter(column);
    const rowPart = (row + 1).toString();
    return `${columnAbsolute ? '$' : ''}${colPart}${rowAbsolute ? '$' : ''}${rowPart}`;
}

/**
 * Check if a string is a valid cell address within the defined grid limits.
 * @param {string} address - The string to check.
 * @returns {boolean} True if valid.
 */
export function isValidCellAddress(address) {
    try {
        parseCellAddress(address);
        return true;
    } catch {
        return false;
    }
}

/**
 * Convert a 0-based column index to Excel-style letters (0->A, 1->B, ..., 25->Z).
 * @param {number} column - 0-based column index.
```

```
 * @returns {string} Column letters.
 */
export function columnToLetter(column) {
    if (column < 0 || column >= MAX_COLUMNS) {
        throw new Error(`Column index out of bounds: ${column}`);
    }
    // For this project (MAX_COLUMNS=26), a simple one-letter mapping suffices.
    return String.fromCharCode(65 + column); // 65 is 'A'
}

/**
 * Convert Excel-style column letters to a 0-based index.
 * @param {string} letters - One or more uppercase letters (e.g., "A", "AB").
 * @returns {number} 0-based column index.
 */
export function letterToColumn(letters) {
    let result = 0;
    for (let i = 0; i < letters.length; i++) {
        const charCode = letters.charCodeAt(i) - 65; // 'A' is 65
        if (charCode < 0 || charCode >= 26) {
            throw new Error(`Invalid column letter: "${letters[i]}" in "${letters}"`);
        }
        result = result * 26 + charCode + 1;
    }
    return result - 1; // Convert from 1-based to 0-based
}

// Export constants as defined in naming conventions
export const MAX_ROWS = 1000;
export const MAX_COLUMNS = 26;
export const VIEWPORT_ROWS = 20;
export const VIEWPORT_COLUMNS = 10;
```

2. **Core Data Structure Skeletons (`src/model/cell.js`, `src/state/spreadsheet.js`):** Define the foundational types.

JAVASCRIPT

```
// src/model/cell.js

export class Cell {

    /**

     * @param {any} rawValue - The raw input value (string, number, or formula string starting with '=')

     */

    constructor(rawValue = '') {

        // TODO 1: Store the raw value.

        // TODO 2: Determine if rawValue is a formula (starts with '=').

        // TODO 3: If it's a formula, store it in this.formula. The evaluated result will go in this.value.

        // TODO 4: If it's not a formula, store it directly in this.value.

        // TODO 5: Initialize this.dependencies as a new Set to hold cell addresses (strings) this cell's
formula depends on.

        // TODO 6: Initialize this.format as an empty object (to later hold formatting options).

        this.value = null;

        this.formula = null;

        this.dependencies = new Set();

        this.format = {};

    }


    // ... methods for evaluation will be added in later milestones

}
```

```javascript
// src/state/spreadsheet.js

import { DependencyGraph } from '../model/dependency-graph/graph.js';


export class Spreadsheet {

    constructor() {

        // TODO 1: Initialize this.grid as a 2D structure (e.g., nested Map or array) sized MAX_ROWS x MAX_COLUMNS.

        // TODO 2: Populate it with empty Cell objects.

        // TODO 3: Initialize this.graph as a new DependencyGraph().

        // TODO 4: Initialize this.selection as { row: 0, col: 0 } (selects cell A1 by default).

        // TODO 5: Initialize a history manager for undo/redo (to be implemented in Milestone 4).

        this.grid = new Map(); // Map<rowIndex, Map<colIndex, Cell>>

        this.graph = new DependencyGraph();

        this.selection = { row: 0, col: 0 };

        // this.history = new History();

    }


    getCell(row, col) {

        // TODO: Return the Cell object at the given row and column.

        // Hint: Check if the row Map exists, then if the cell exists.

    }


    setCellValue(row, col, rawValue) {

        // TODO: This is the main entry point for editing a cell.

        // TODO 1: Create a new Cell with the rawValue.

        // TODO 2: Update the grid at (row, col).

        // TODO 3: Update the dependency graph (remove old edges, parse formula for new dependencies, add new edges).

        // TODO 4: Trigger a recalculation of all affected cells (starting from this cell's dependents).

        // TODO 5: For Milestone 4: Record this change in the history stack for undo/redo.

    }

    // ... more methods

}
```

Milestone Checkpoint: Project Setup

After setting up the basic structure and helper modules, verify your foundation:

1. **Run the project:** Use a simple static server (e.g., `npx serve public`) and open the browser. You should see a blank page (UI not built yet).
2. **Test address utilities:** Open the browser's Developer Console and try the imported functions.

```javascript
import { parseCellAddress, toCellAddress } from './src/model/cell-address.js';

console.log(parseCellAddress('B5')); // Should log { row: 4, column: 1, ... }

console.log(toCellAddress(4, 1)); // Should log "B5"
```

3. **Check for errors:** Ensure there are no syntax errors in the browser's console. The basic skeleton should load without throwing exceptions.

Signs of Trouble:

- **"Uncaught SyntaxError: Cannot use import statement outside a module"**: You need to add `type="module"` to your `<script>` tag in `index.html`: `<script type="module" src="src/index.js"></script>`.
- **Functions are `undefined`**: Check that your import paths are correct relative to the file executing the code.

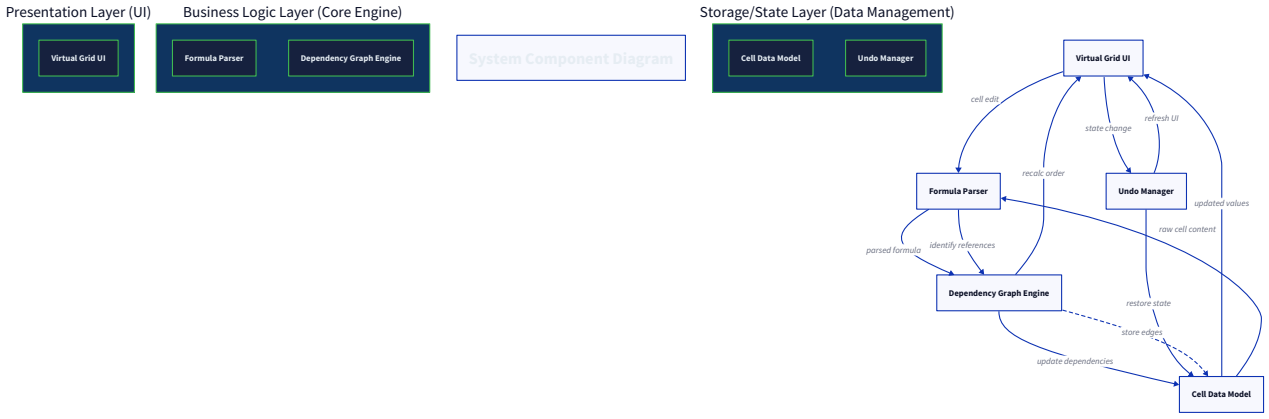
High-Level Architecture

> **Milestone(s):** All (provides the architectural foundation for all four milestones)

This section presents the overarching architectural design that will enable our spreadsheet application to function efficiently while remaining maintainable and extensible. The primary challenge in spreadsheet architecture is managing the **reactive dataflow** where any cell change must trigger a cascade of recalculations in dependent cells, all while maintaining smooth user interaction. Our solution organizes the system into three distinct layers, each with clear responsibilities and clean interfaces.

Three-Layer Architecture Overview

Think of building a spreadsheet as constructing a multi-story factory: the **Storage/State Layer** is the warehouse and inventory system, the **Business Logic Layer** is the factory floor where calculations happen, and the **Presentation Layer** is the storefront where customers interact with the final product. This separation ensures that changes in one area (like updating the UI) don't disrupt the core calculation engine.



The architecture consists of three primary layers with unidirectional data flow:

1. **Presentation Layer (UI)**: Handles all user interaction and visual rendering
2. **Business Logic Layer (Core Engine)**: Processes formulas, manages dependencies, and executes calculations
3. **Storage/State Layer (Data Management)**: Stores cell data, manages history, and handles persistence

This layered approach follows the **separation of concerns** principle: each layer has a specific domain of responsibility and communicates with adjacent layers through well-defined interfaces. The flow of control typically moves from the Presentation Layer (user action) to the Business Logic Layer (processing) to the Storage/State Layer (data update), and then back through the layers to update the UI.

> **Key Insight:** The dependency graph in the Business Logic Layer acts as the system's nervous system—it knows exactly which cells need to wake up (recalculate) when any cell changes, ensuring the entire spreadsheet stays consistent without unnecessary work.

Data Flow Pattern: The architecture implements a variation of the Model-View-Controller (MVC) pattern tailored for reactive systems:

- **Model**: Storage/State Layer + Business Logic Layer (data and business rules)
- **View**: Presentation Layer (visual representation)
- **Controller**: Event handlers in the Presentation Layer that dispatch to the Business Logic Layer

Architecture Decision: Three-Layer Separation

> **Decision: Layered Architecture with Clear Separation**
>
> - **Context**: We need to support complex spreadsheet functionality (formulas, dependencies, UI) while maintaining code that is testable, debuggable, and evolvable over time. A monolithic architecture would become unmanageable as features accumulate.
> - **Options Considered**:
>   1. **Monolithic component** (all code in one module)
>   2. **Two-layer architecture** (UI + everything else)
>   3. **Three-layer architecture** (UI, Business Logic, Data Storage)
> - **Decision**: Three-layer architecture with the specific layer breakdown described above.
> - **Rationale**:
>   - **Testability**: Business logic can be unit tested without UI dependencies
>   - **Maintainability**: Changes to rendering don't affect calculation logic
>   - **Performance**: Virtual scrolling in the UI layer can be optimized independently
>   - **Extensibility**: New features (like undo/redo or CSV export) can be added to appropriate layers without disrupting others
> - **Consequences**:
>   - Adds some complexity in managing layer boundaries
>   - Requires careful design of interfaces between layers
>   - Enables parallel development of different components

Option	Pros	Cons	Why Not Chosen
Monolithic component	Simple to understand initially; No cross-layer communication overhead	Becomes unmanageable beyond ~500 lines; Hard to test; Changes affect everything	Doesn't scale to the complexity of a spreadsheet engine
Two-layer (UI + everything)	Less communication overhead than three layers; Still some separation	Business logic and data storage become entangled; Hard to add features like undo/redo later	Lacks the clean separation needed for advanced features
Three-layer architecture	**Clear separation of concerns; Testable; Extensible; Parallel development possible**	**Requires careful interface design; Slight overhead in layer communication**	**Best balance of structure and flexibility for our needs**

Component Map and Responsibilities

Each layer contains specific components with well-defined responsibilities. The following table maps each component to its layer, primary responsibility, and key data structures:

Layer	Component	Primary Responsibility	Key Data Structures Managed	Interfaces Exposed
Presentation Layer	Virtual Grid Renderer	Renders visible cells using DOM recycling; handles cell selection and editing UI	Viewport state (visible rows/columns); Cell DOM element cache	`renderViewport()`, `enterEditMode(cellAddress)`, `updateCellDisplay(cellAddress, value)`
	Event Handler Coordinator	Routes user events (clicks, keyboard, scroll) to appropriate handlers; manages focus	Selection state; Keyboard shortcuts map	`handleCellClick(event)`, `handleKeyDown(event)`, `handleScroll(event)`
Business Logic Layer	Formula Parser & Evaluator	Converts formula strings to executable expressions; evaluates formulas with current cell values	AST nodes; Function registry; Evaluation context	`parseFormula(formulaString)`, `evaluate(ast, context)`, `resolveReference(cellAddress)`
	Dependency Graph Engine	Tracks cell dependencies; determines recalculation order; detects circular references	`DependencyGraph` (nodes, edges); Dirty cell set; Topological order	`addDependency(dependent, dependency)`, `getTopologicalOrder()`, `detectCircularReference()`
	Recalculation Scheduler	Coordinates recalculation when cells change; optimizes by only recalculating dirty cells	Dirty cell queue; In-progress recalculation flag	`scheduleRecalculation(changedCellAddress)`, `recalculateCell(cellAddress)`
Storage/State Layer	Cell Data Model	Stores cell values, formulas, and formatting; provides atomic updates to cell data	`Spreadsheet` (grid of `Cell` objects); Cell metadata	`getCell(row, col)`, `setCellValue(row, col, value)`, `setCellFormula(row, col, formula)`

Layer	Component	Primary Responsibility	Key Data Structures Managed	Interfaces Exposed
	Undo/Redo Manager	Tracks cell changes for reversal; manages undo/redo stacks	Command stack; Inverse command storage	`executeCommand(command)`, `undo()`, `redo()`, `canUndo()`, `canRedo()`
	Format & Style Manager	Handles cell formatting (number formats, alignment, colors)	Format rules cache; Style mapping	`applyFormat(cellAddress, format)`, `getDisplayValue(cellAddress)`
	Import/Export Handler	Converts between spreadsheet data and external formats (CSV, etc.)	Serialization buffer; Parsing state	`exportToCSV()`, `importFromCSV(csvText)`

Component Interaction Patterns:

1. **Cell Edit Flow**:
   - User types in cell → Event Handler captures input
   - Cell Data Model updates raw value/formula
   - Formula Parser parses if formula present
   - Dependency Graph Engine updates edges
   - Recalculation Scheduler determines affected cells
   - Cell values update in Data Model
   - Virtual Grid Renderer updates display
2. **Scroll Interaction**:
   - User scrolls → Event Handler detects scroll
   - Virtual Grid Renderer computes new viewport
   - Renderer reuses DOM elements for new visible cells
   - Cell Data Model provides data for newly visible cells
3. **Undo/Redo Flow**:
   - User triggers undo → Undo/Redo Manager pops command
   - Command executes inverse operation
   - Cell Data Model updates affected cells
   - Recalculation triggered as needed
   - UI updates to reflect changes

Mental Model: The Spreadsheet as a Reactive Assembly Line

Imagine an automobile factory assembly line. The **Cell Data Model** is the inventory system tracking all parts. The **Dependency Graph Engine** is the production schedule that knows Car needs Engine before it can need Wheels. The **Formula Parser** is the quality control station that checks if instructions make sense. The **Virtual Grid Renderer** is the showroom display that only shows finished cars (visible cells), not the entire factory floor. When a supplier delivers new seats (cell change), the schedule (dependency graph) tells which cars (dependent cells) need to be updated, and only those cars return to the assembly line for updates.

Recommended File and Module Structure

Organizing code into a clear directory structure from the beginning prevents "spaghetti code" and makes the architecture tangible. Below is the recommended structure for a TypeScript/JavaScript implementation:

```
spreadsheet-engine/
├── package.json
├── tsconfig.json (or jsconfig.json)
├── index.html
├── src/
│   ├── index.ts (or index.js)          # Application entry point
│   ├── main.css                        # Global styles
│   │
│   ├── presentation/                   # Presentation Layer
│   │   ├── grid/
│   │   │   ├── VirtualGrid.ts          # Main grid component with virtualization
│   │   │   ├── ViewportManager.ts      # Calculates visible cells
│   │   │   ├── CellRenderer.ts         # Renders individual cell DOM elements
│   │   │   ├── SelectionManager.ts     # Handles cell selection state
│   │   │   └── index.ts                # Public exports
│   │   │
│   │   └── events/
│   │       ├── EventCoordinator.ts     # Central event dispatcher
│   │       ├── KeyboardHandler.ts      # Keyboard shortcuts
│   │       ├── MouseHandler.ts         # Click/double-click handling
│   │       └── index.ts
│   │
│   ├── business-logic/                 # Business Logic Layer
│   │   ├── formula/
│   │   │   ├── Parser.ts               # Formula string to AST
│   │   │   ├── Tokenizer.ts            # Lexical analysis
│   │   │   ├── AST/                    # Abstract Syntax Tree types
│   │   │   │   ├── nodes.ts            # AST node definitions
│   │   │   │   └── evaluator.ts        # AST evaluation engine
│   │   │   ├── functions.ts            # Built-in functions (SUM, AVG, etc.)
│   │   │   └── reference.ts            # Cell reference parsing utilities
│   │   │
│   │   └── dependency/
│   │       ├── DependencyGraph.ts      # Graph structure and algorithms
│   │       ├── TopologicalSorter.ts    # Topological sort implementation
│   │       ├── CycleDetector.ts        # Circular reference detection
│   │       └── RecalculationScheduler.ts # Coordinates cell updates
│   │
│   ├── storage/                        # Storage/State Layer
│   │   ├── model/
│   │   │   ├── Spreadsheet.ts          # Main data model class
│   │   │   ├── Cell.ts                 # Cell data structure
│   │   │   └── Grid.ts                 # 2D grid management
│   │   │
│   │   ├── history/
│   │   │   ├── UndoRedoManager.ts      # Command pattern implementation
│   │   │   ├── commands.ts             # Concrete command classes
│   │   │   └── CommandHistory.ts       # Stack management
│   │   │
│   │   ├── format/
│   │   │   ├── CellFormatter.ts        # Number/date formatting
│   │   │   └── StyleManager.ts         # Visual styles (colors, alignment)
│   │   │
│   │   └── import-export/
│   │       ├── CSVExporter.ts
│   │       └── CSVImporter.ts
│   │
│   ├── utils/                          # Shared utilities
│   │   ├── cellAddress.ts              # parseCellAddress, toCellAddress, etc.
│   │   ├── math.ts                     # Math utilities
│   │   └── constants.ts                # MAX_ROWS, MAX_COLUMNS, etc.
│   │
│   └── types/                          # TypeScript type definitions
│       ├── core.ts                     # Core interfaces (Cell, Spreadsheet, etc.)
```

```
│       └── events.ts                    # Event-related types
│
└── tests/                               # Test suites
    ├── unit/
    │   ├── formula/
    │   ├── dependency/
    │   └── model/
    └── integration/
        └── recalculation-flow.test.ts
```

Module Dependency Rules:

1. **Presentation Layer** can import from **Business Logic Layer** and **Storage Layer**, but not vice versa
2. **Business Logic Layer** can import from **Storage Layer**, but not from Presentation Layer
3. **Storage Layer** should have no dependencies on other layers (lowest layer)
4. **Utils** and **Types** can be imported by any layer (shared infrastructure)

File Naming Conventions:

- Use PascalCase for classes and TypeScript files: `VirtualGrid.ts`
- Use camelCase for utility functions and instances: `parseCellAddress`
- Use .ts extension for TypeScript, .js for JavaScript
- Test files match source files: `VirtualGrid.test.ts` alongside `VirtualGrid.ts`

Architecture Decision: File Organization by Layer

> **Decision: Layer-Centric Directory Structure**
>
> - **Context**: We need to make the architectural boundaries visible in the codebase to prevent accidental coupling between unrelated components. Developers should intuitively understand what belongs together.
> - **Options Considered**:
>   1. **Feature-based grouping** (all spreadsheet code together, all formula code together)
>   2. **Type-based grouping** (all components together, all utilities together)
>   3. **Layer-based grouping** (presentation/, business-logic/, storage/)
> - **Decision**: Layer-based directory structure with subdirectories for components within each layer.
> - **Rationale**:
>   - **Clarity**: Immediately communicates architectural separation
>   - **Enforcement**: Makes it harder to create circular dependencies between layers
>   - **Navigation**: Developers working on UI know to look in presentation/
>   - **Scalability**: New features can be added to appropriate layers without restructuring
> - **Consequences**:
>   - Some related code (e.g., formula parsing and evaluation) is separated into different directories within the same layer
>   - Requires discipline to maintain layer boundaries
>   - Import paths become slightly longer

Option	Pros	Cons	Why Not Chosen
Feature-based grouping	All code for a feature is together; Easy to find related code	Encourages tight coupling; Hard to see architectural layers; Business logic mixes with UI	Violates separation of concerns; Makes testing harder
Type-based grouping	Consistent pattern (all components together); Easy to find "all utilities"	No architectural guidance; UI components mix with core logic	Doesn't help enforce architectural boundaries
Layer-based grouping	**Clear architectural boundaries; Natural dependency direction; Scalable structure**	**Some related code separated; Longer import paths**	**Best enforces the three-layer architecture we've designed**

Entry Point and Initialization: The main application entry point (`src/index.ts`) is responsible for wiring together all components:

1. Instantiate the Storage Layer components (Spreadsheet model, UndoRedoManager)
2. Instantiate the Business Logic Layer components (Parser, DependencyGraph)
3. Instantiate the Presentation Layer components (VirtualGrid, EventCoordinator)
4. Connect the layers: pass model references to business logic, pass business logic references to UI
5. Initialize the grid with initial data and attach to DOM

This bootstrapping follows the **dependency injection** pattern, where higher layers receive their dependencies from lower layers, ensuring clean separation and testability.

Common Pitfalls: Architecture

⚠️ Pitfall: Business Logic in UI Components

- **Description**: Putting formula evaluation or dependency tracking logic directly in grid rendering code
- **Why it's wrong**: Makes testing nearly impossible; UI changes can break calculations; cannot reuse logic for other purposes (e.g., headless calculation engine)
- **Fix**: Strictly enforce layer boundaries. The Presentation Layer should only delegate to Business Logic Layer methods, not implement them.

⚠️ Pitfall: Bidirectional Dependencies Between Layers

- **Description**: Business Logic Layer importing UI components or Presentation Layer directly accessing Storage Layer internals
- **Why it's wrong**: Creates circular dependencies; makes code brittle; violates the unidirectional data flow principle
- **Fix**: Use the dependency injection pattern. Higher layers should only depend on abstractions (interfaces) of lower layers, not concrete implementations.

⚠️ Pitfall: God Object in Storage Layer

- **Description**: Putting all data (cells, history, formats) in one massive `Spreadsheet` class with hundreds of methods
- **Why it's wrong**: Violates single responsibility principle; hard to test; changes affect everything
- **Fix**: Delegate responsibilities to specialized managers (UndoRedoManager for history, CellFormatter for formatting) that the main Spreadsheet class coordinates.

⚠️ Pitfall: Inline Constants Everywhere

- **Description**: Hardcoding `MAX_ROWS = 1000` in multiple component files
- **Why it's wrong**: Changing the constant requires updating many files; inconsistent values cause subtle bugs
- **Fix**: Centralize constants in `src/utils/constants.ts` and import them where needed.

Implementation Guidance

Technology Recommendations Table:

Component	Simple Option	Advanced Option	Recommendation for Learning
UI Framework	Vanilla DOM API	React/Vue/Angular	**Vanilla DOM** (teaches fundamentals without framework magic)
Build Tool	None (ES modules in browser)	Webpack/Vite	**ES modules** (simplest setup; modern browsers support)
Testing	Manual testing in browser	Jest + jsdom	**Jest** (industry standard; good for unit testing logic)
Type Safety	Plain JavaScript	TypeScript	**TypeScript** (catches errors early; excellent for complex data structures)
State Management	Custom event system	Redux/MobX	**Custom event system** (understand fundamentals before libraries)

Recommended Starter File Structure Implementation:

Create the following minimal structure to begin development:

```
spreadsheet-engine/
├── index.html
├── src/
│   ├── index.ts
│   ├── presentation/
│   │   └── grid/
│   │       └── VirtualGrid.ts
│   ├── business-logic/
│   │   └── formula/
│   │       └── reference.ts
│   ├── storage/
│   │   └── model/
│   │       └── Spreadsheet.ts
│   └── utils/
│       ├── cellAddress.ts
│       └── constants.ts
```

Infrastructure Starter Code:

Below is complete, working code for foundational utilities that are prerequisites but not the core learning goal:

`src/utils/constants.ts` - Centralized Constants:

```typescript
// Spreadsheet dimensions
export const MAX_ROWS = 1000;
export const MAX_COLUMNS = 26;
export const VIEWPORT_ROWS = 20;
export const VIEWPORT_COLUMNS = 10;

// Cell addressing
export const COLUMN_LETTERS = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ';

// Error messages
export const ERRORS = {
  CIRCULAR_REF: '#CIRCULAR!',
  PARSE_ERROR: '#ERROR!',
  REF_ERROR: '#REF!',
  VALUE_ERROR: '#VALUE!'
} as const;
```

`src/utils/cellAddress.ts` - Cell Address Utilities (COMPLETE):

TYPESCRIPT

```typescript
import { COLUMN_LETTERS } from './constants';

export interface ParsedCellReference {
  row: number;           // 0-based row index
  column: number;        // 0-based column index
  rowAbsolute: boolean;  // True if row has $ prefix
  columnAbsolute: boolean; // True if column has $ prefix
}

/**
 * Parse Excel-style cell address into structured format
 * Examples: "A1" -> {row: 0, column: 0, rowAbsolute: false, columnAbsolute: false}
 *           "$B$3" -> {row: 2, column: 1, rowAbsolute: true, columnAbsolute: true}
 *           "C$5" -> {row: 4, column: 2, rowAbsolute: true, columnAbsolute: false}
 */
export function parseCellAddress(address: string): ParsedCellReference {
  // Regular expression to match cell references with optional $ signs
  const match = address.match(/^(\$?)([A-Z]+)(\$?)(\d+)$/);
  if (!match) {
    throw new Error(`Invalid cell address: ${address}`);
  }

  const [, colAbs, colLetters, rowAbs, rowNum] = match;
  const column = letterToColumn(colLetters);
  const row = parseInt(rowNum, 10) - 1; // Convert to 0-based

  if (row < 0 || row >= MAX_ROWS || column < 0 || column >= MAX_COLUMNS) {
    throw new Error(`Cell address out of bounds: ${address}`);
  }

  return {
    row,
    column,
```

```
    rowAbsolute: rowAbs === '$',
    columnAbsolute: colAbs === '$'
  };
}

/**
 * Convert row/column indices to Excel-style cell address string
 */
export function toCellAddress(
  row: number,
  column: number,
  rowAbsolute: boolean = false,
  columnAbsolute: boolean = false
): string {
  if (row < 0 || row >= MAX_ROWS || column < 0 || column >= MAX_COLUMNS) {
    throw new Error(`Cell indices out of bounds: row=${row}, column=${column}`);
  }

  const colPrefix = columnAbsolute ? '$' : '';
  const rowPrefix = rowAbsolute ? '$' : '';

  return `${colPrefix}${columnToLetter(column)}${rowPrefix}${row + 1}`;
}

/**
 * Check if string is a valid cell address
 */
export function isValidCellAddress(address: string): boolean {
  try {
    parseCellAddress(address);
    return true;
  } catch {
    return false;
```

```
  }
}

/**
 * Convert 0-based column index to Excel column letters
 * 0 -> "A", 1 -> "B", ..., 25 -> "Z"
 */
export function columnToLetter(column: number): string {
  if (column < 0 || column >= MAX_COLUMNS) {
    throw new Error(`Column index out of bounds: ${column}`);
  }

  // For columns 0-25 (A-Z), simple lookup works
  return COLUMN_LETTERS[column];
}

/**
 * Convert Excel column letters to 0-based index
 * "A" -> 0, "B" -> 1, ..., "Z" -> 25
 */
export function letterToColumn(letters: string): number {
  letters = letters.toUpperCase();
  let result = 0;

  for (let i = 0; i < letters.length; i++) {
    const char = letters.charCodeAt(i) - 65; // 'A' = 65
    if (char < 0 || char > 25) {
      throw new Error(`Invalid column letter: ${letters[i]}`);
    }
    result = result * 26 + char;
  }

  // For this project, we only support single letters (A-Z)
```

```
  if (result >= MAX_COLUMNS) {
    throw new Error(`Column ${letters} exceeds maximum column Z`);
  }

  return result;
}
```

`src/storage/model/Cell.ts` - Core Cell Data Structure (COMPLETE):

TYPESCRIPT

```typescript
export interface CellFormat {
  numberFormat?: 'currency' | 'percent' | 'number' | 'general';
  decimalPlaces?: number;
  alignment?: 'left' | 'center' | 'right';
  bold?: boolean;
  italic?: boolean;
}

export interface Cell {
  // The calculated value displayed in the cell
  value: any;

  // The raw formula string (if any), e.g., "=A1+B2"
  formula: string | null;

  // Formatting options for display
  format: CellFormat;

  // Set of cell addresses this cell depends on (for dependency tracking)
  dependencies: Set<string>;

  // Cached parsed AST for performance (optional optimization)
  cachedAST?: any;
}

/**
 * Create a new empty cell with default values
 */
export function createEmptyCell(): Cell {
  return {
    value: '',
    formula: null,
    format: {},
```

```
    dependencies: new Set<string>()
  };
}

/**
 * Create a cell with a numeric value
 */
export function createNumberCell(value: number): Cell {
  return {
    value,
    formula: null,
    format: { numberFormat: 'number' },
    dependencies: new Set<string>()
  };
}

/**
 * Create a cell with a formula
 */
export function createFormulaCell(formula: string): Cell {
  return {
    value: null, // Will be calculated
    formula,
    format: {},
    dependencies: new Set<string>()
  };
}
```

Core Logic Skeleton Code:

`src/storage/model/Spreadsheet.ts` **- Main Data Model (TODO skeleton):**

TYPESCRIPT

```typescript
import { MAX_ROWS, MAX_COLUMNS } from '../../utils/constants';
import { Cell, createEmptyCell } from './Cell';
import { toCellAddress, parseCellAddress } from '../../utils/cellAddress';

export class Spreadsheet {
  // 2D grid of cells using nested Maps for sparse storage
  private grid: Map<number, Map<number, Cell>>;

  // Current selection (active cell)
  public selection: { row: number; col: number };

  constructor() {
    this.grid = new Map();
    this.selection = { row: 0, col: 0 };

    // Initialize the grid structure (empty, sparse representation)
    // TODO 1: Initialize grid as an empty Map (sparse representation)
    // TODO 2: Set initial selection to A1 (row 0, col 0)
  }

  /**
   * Get cell at specified row and column (0-based indices)
   * Returns a new empty cell if none exists at that location
   */
  getCell(row: number, col: number): Cell {
    // TODO 3: Check if row exists in grid Map
    // TODO 4: If row exists, check if column exists in that row's Map
    // TODO 5: Return existing cell or create empty cell if not found
    // TODO 6: Consider cloning the cell to prevent accidental mutation
    return createEmptyCell(); // Placeholder
  }

  /**
```

```
 * Set raw value for a cell (not a formula)
 */
setCellValue(row: number, col: number, value: any): void {
  // TODO 7: Get or create cell at position
  // TODO 8: Update cell's value property
  // TODO 9: Clear any existing formula (since we're setting raw value)
  // TODO 10: Clear dependencies (raw values don't have dependencies)
  // TODO 11: Consider returning the cell address for dependency graph updates
}

/**
 * Set formula for a cell
 */
setCellFormula(row: number, col: number, formula: string): void {
  // TODO 12: Get or create cell at position
  // TODO 13: Update cell's formula property
  // TODO 14: Clear the cached value (will be recalculated)
  // TODO 15: Return the cell address for later parsing
}

/**
 * Get cell by address string (e.g., "A1")
 */
getCellByAddress(address: string): Cell {
  // TODO 16: Parse address using parseCellAddress utility
  // TODO 17: Call getCell with parsed row and column
  return createEmptyCell(); // Placeholder
}

/**
 * Get all cells that have data (non-empty iterator for efficient traversal)
 */
```

```
  *getAllCells(): Generator<{row: number; col: number; cell: Cell}> {
    // TODO 18: Iterate through rows in grid Map
    // TODO 19: For each row, iterate through columns in that row's Map
    // TODO 20: Yield each cell with its coordinates
  }
}
```

`src/business-logic/dependency/DependencyGraph.ts` - Dependency Graph Skeleton:

TYPESCRIPT

```typescript
import { Cell } from '../../storage/model/Cell';

export class DependencyGraph {
  // Map from cell address to Cell object
  public nodes: Map<string, Cell>;

  // Map from cell address to Set of addresses it depends on (edges: node -> dependencies)
  public edges: Map<string, Set<string>>;

  constructor() {
    this.nodes = new Map();
    this.edges = new Map();
  }

  /**
   * Add a dependency relationship: dependentCell depends on dependencyCell
   */
  addDependency(dependentAddress: string, dependencyAddress: string): void {
    // TODO 1: Ensure both nodes exist in the nodes Map
    // TODO 2: Get or create edge set for dependentAddress
    // TODO 3: Add dependencyAddress to the edge set
    // TODO 4: Update the dependent cell's dependencies property
    // TODO 5: Check for circular reference (stretch goal for later)
  }

  /**
   * Remove a dependency relationship
   */
  removeDependency(dependentAddress: string, dependencyAddress: string): void {
    // TODO 6: Get edge set for dependentAddress
    // TODO 7: If set exists, delete dependencyAddress from it
    // TODO 8: Update the cell's dependencies property
  }
```

```
  /**
   * Get all cells that directly depend on the given cell
   */
  getDependents(cellAddress: string): Set<string> {
    // TODO 9: Iterate through all edges to find cells that have cellAddress in their dependency set
    // TODO 10: Return set of dependent addresses
    return new Set(); // Placeholder
  }

  /**
   * Get topological order for recalculation (dependencies before dependents)
   */
  getTopologicalOrder(changedCell: string): string[] {
    // TODO 11: Perform depth-first search starting from changedCell
    // TODO 12: Collect all reachable cells (transitive dependents)
    // TODO 13: Sort cells so dependencies come before dependents
    // TODO 14: Handle cycles (return empty array or throw error)
    return []; // Placeholder
  }

  /**
   * Detect if adding an edge would create a circular reference
   */
  detectCircularReference(startAddress: string, targetAddress: string): boolean {
    // TODO 15: Perform depth-first search from targetAddress
    // TODO 16: Check if we reach startAddress (would create cycle)
    // TODO 17: Return true if cycle detected
    return false; // Placeholder
  }
}
```

Language-Specific Hints for TypeScript/JavaScript:

1. **Use Maps for Sparse Grids**: Since most cells are empty, use `Map<number, Map<number, Cell>>` instead of a 2D array to save memory.
2. **TypeScript Generics for Safety**: Define interfaces for all data structures to catch type errors at compile time.
3. **ES6 Modules**: Use `import` / `export` syntax for clean module boundaries. Configure `tsconfig.json` with `"module": "esnext"`.
4. **Event Delegation for Performance**: Attach a single event listener to the grid container instead of individual cell listeners.
5. **RequestAnimationFrame for Smooth Scrolling**: Use `window.requestAnimationFrame()` for scroll-induced redraws to avoid jank.

Milestone Checkpoint for Architecture Setup:

After setting up the basic architecture:

1. **Verify**: Run `npm run build` (if using TypeScript) and check for compilation errors
2. **Test**: Create a simple HTML file that imports your entry point and check browser console for errors
3. **Structure Validation**: Ensure you can import from `presentation/` to `business-logic/` to `storage/` but not vice versa
4. **Expected Behavior**: The browser should show a blank page without JavaScript errors
5. **Troubleshooting**: If you see "Cannot find module" errors, check your import paths and TypeScript configuration

Next Steps: Begin with Milestone 1 by implementing the `VirtualGrid` component in the presentation layer, using the `Spreadsheet` class from the storage layer as the data source.

Data Model

> **Milestone(s):** Milestone 1, 2, 3, 4 (The data model underpins all milestones and represents the fundamental structures that every component interacts with)

The data model forms the bedrock of the spreadsheet application, defining how we represent, store, and connect the core spreadsheet concepts. Think of this as the DNA of your spreadsheet—every feature, from simple cell rendering to complex formula recalculation, will rely on these fundamental data structures. This section provides a comprehensive blueprint for the core types that will persist across user sessions, enable efficient dependency tracking, and support the reactive dataflow that makes spreadsheets so powerful.


Data Model Class Diagram
Core Data Model
Spreadsheet
+ grid Grid
+ dependency_graph DependencyGraph
contains 1
Grid
+ rows int
+ cols int
+ cells Cell[][]
has 1
contains many
Cell
+ value any
+ formula string
+ format string
+ dependencies string[]
references many
Dependency Tracking
DependencyGraph
+ nodes Node[]
+ edges Edge[]
contains many
contains many
Node
+ cell_reference string
+ depends_on Edge[]
+ dependents Edge[]
references many
references many
from 1
to 1
Edge
+ source_node Node
+ target_node Node

Cell Data Structure: Value, Formula, and Metadata

Mental Model: A Cell is a Tiny Computer with Memory and Processing Instructions

Imagine each cell as a miniature computer that can operate in two modes. In "memory mode," it simply stores a static value like `42` or `"Hello"`. In "processing mode," it contains instructions (a formula) that tell it how to compute a value by reading from other cells' memory. The cell's current display is always the result of whichever mode it's in: either the stored value directly, or the computed result of executing its instructions. This duality—storing versus computing—is central to the spreadsheet paradigm and must be captured in our data structure.

The `Cell` type represents this fundamental unit of spreadsheet data. It must track not only what the cell displays (its current value) but also how that value was derived (its formula, if any), how it should appear (formatting), and—critically—which other cells' values it needs to read to compute its own result (its dependencies). This last piece is what enables the reactive dataflow: when a dependency changes, we know which cells need recomputation.

Cell Data Structure Table

Field Name	Type	Description
`value`	`any` (string | number | boolean | Error)	The current computed or stored value of the cell. This is what gets displayed in the grid. For formula cells, this is the result of the most recent evaluation. For literal cells, this is the raw input value. Errors are special values (like `#REF!` or `#DIV/0!`) that propagate through dependencies.
`formula`	`string`	The raw formula string as entered by the user, including the leading `=` character. Empty string indicates a non-formula cell (literal value). This preserves the exact user input for editing and display in the formula bar.
`format`	`CellFormat` object	An object containing all visual formatting metadata for the cell: number formatting, alignment, font styles, etc. This controls how the `value` is presented but doesn't affect the underlying computation.
`dependencies`	`Set<string>`	A set of cell addresses (like `"A1"`, `"B2:C5"`) that this cell's formula references directly. These are the immediate inputs needed to compute this cell's value. We store these addresses as strings for simplicity and easy serialization. The presence of this field enables the dependency graph to know which edges exist from this cell to its dependencies.

CellFormat Data Structure Table

Field Name	Type	Description
`numberFormat`	`string`	Format pattern for numeric values (e.g., `"currency"`, `"percentage"`, `"0.00"`, `"mm/dd/yyyy"`). Determines how numbers are converted to display strings. Default: `"general"`.
`decimalPlaces`	`number`	Number of decimal places to show (for numeric formats). Overrides format pattern precision when specified. Default: `null` (use format pattern default).
`alignment`	`string`	Horizontal alignment: `"left"`, `"center"`, `"right"`. Default: `"left"` for text, `"right"` for numbers.
`bold`	`boolean`	Whether text should be bold. Default: `false`.
`italic`	`boolean`	Whether text should be italic. Default: `false`.

ParsedCellReference Data Structure Table

Field Name	Type	Description
`row`	`number`	0-based row index (0 = row 1, 99 = row 100). This allows direct indexing into the grid data structure.
`column`	`number`	0-based column index (0 = column A, 25 = column Z). Matches the internal grid representation.
`rowAbsolute`	`boolean`	Whether the row part is absolute (prefixed with `$`). Affects reference adjustment during copy/paste operations.
`columnAbsolute`	`boolean`	Whether the column part is absolute (prefixed with `$`). Affects reference adjustment during copy/paste operations.

Decision: Using `Set<string>` for Cell Dependencies

- **Context**: We need to track which cells a formula references to build the dependency graph. References could be single cells (`A1`) or ranges (`A1:A10`), and we need efficient membership testing and deduplication.
- **Options Considered**:
  1. **Array of strings**: Simple but O(n) lookups, duplicate entries possible.
  2. **Set of `ParsedCellReference` objects**: Type-safe but requires parsing on every operation, complicates range handling.
  3. **Set of strings**: Simple string addresses, works for ranges, efficient O(1) lookups via native Set.
- **Decision**: Use `Set<string>` to store cell addresses as strings (e.g., `"A1"` , `"B2:C5"`).
- **Rationale**: String addresses are easy to serialize/deserialize, work for both single cells and ranges, and leverage JavaScript's native `Set` for O(1) operations. The slight loss of type safety is acceptable because we validate addresses when they're parsed from formulas.
- **Consequences**: We must ensure address strings are normalized (always uppercase, no spaces) to avoid duplicates. Range references will be stored as single strings rather than expanded into individual cells, which saves memory but requires special handling during dependency graph traversal.

Option	Pros	Cons	Chosen?
Array of strings	Simple, ordered	Slow lookups, duplicates possible	❌
Set of `ParsedCellReference`	Type-safe, structured	Overhead parsing, doesn't handle ranges well	❌
Set of strings	**Fast lookups, handles ranges, serializable**	**Less type safety, requires normalization**	✅

Cell Lifecycle Example

Consider cell `C3` containing the formula `=A1+B2` . When the user enters this formula:

1. The formula string `"=A1+B2"` is stored in the `formula` field.
2. The parser extracts references `"A1"` and `"B2"` , which are added to the `dependencies` set.
3. The formula is evaluated by fetching values from cells `A1` and `B2` , computing their sum, and storing the result in `value` .
4. If either `A1` or `B2` changes later, the dependency graph uses the `dependencies` set to know that `C3` needs reevaluation.

Common Pitfalls: Cell Data Structure

⚠️ Pitfall: Storing only the computed value without the original formula

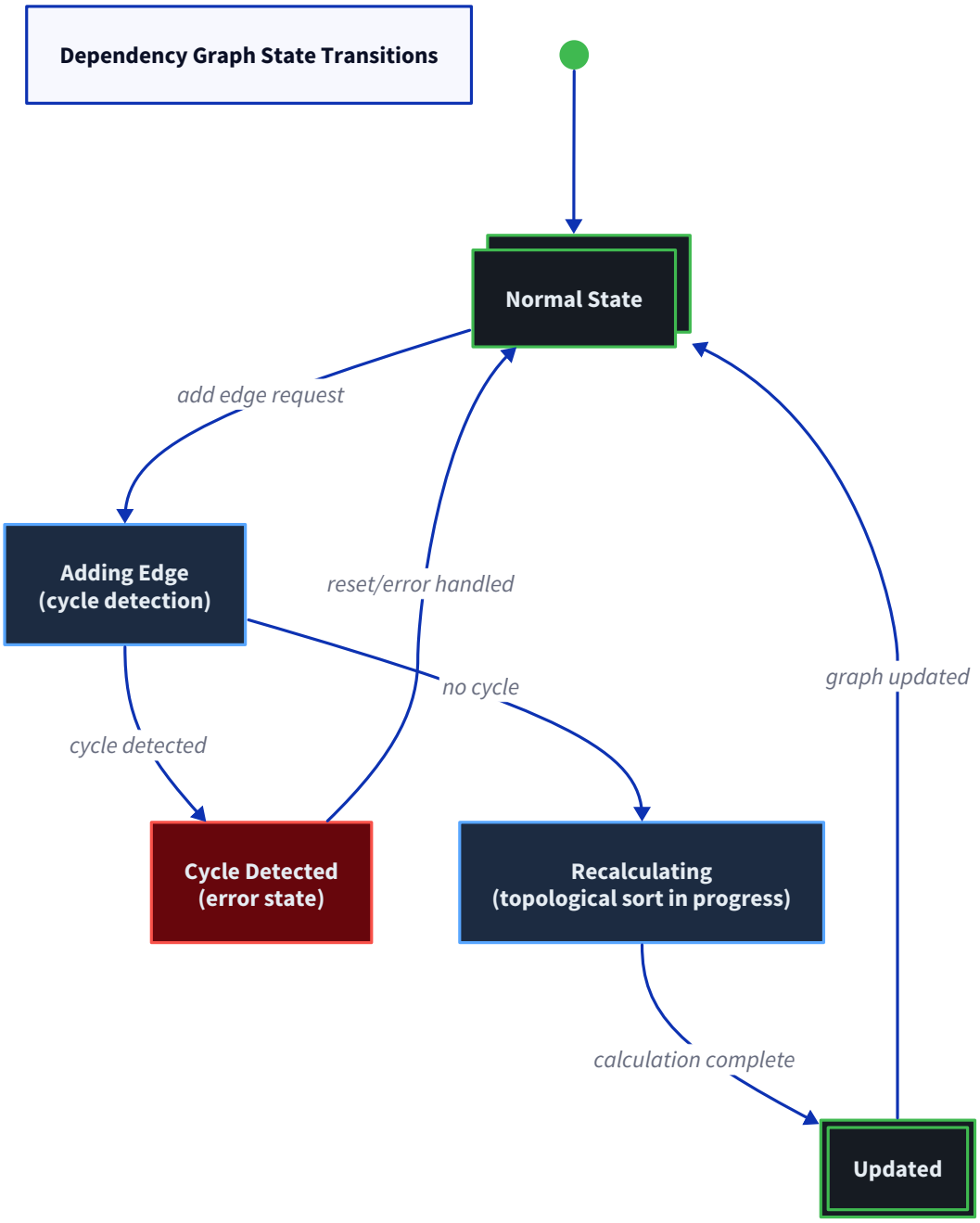
- **Description**: Only saving the current `value` and discarding the `formula` string after evaluation.
- **Why it's wrong**: The user cannot edit the formula later. The formula bar would show the computed value instead of the formula that produced it, breaking the fundamental spreadsheet editing model.
- **Fix**: Always preserve the raw formula string in the `formula` field, even after evaluation. Treat the `value` field as a cache that can be recomputed from the formula when needed.

⚠️ **Pitfall: Using the same field for raw input and computed value**

- **Description**: Having a single `content` field that stores either the literal input or formula result.
- **Why it's wrong**: Makes it impossible to distinguish between a literal value `"=A1"` (the text string) and a formula `=A1` (a reference to cell A1). Also prevents proper formula re-evaluation when dependencies change.
- **Fix**: Use separate `value` and `formula` fields. If `formula` is non-empty, the cell is a formula cell and `value` is its computed result. If `formula` is empty, `value` is a literal value.

Dependency Graph: Nodes, Edges, and Adjacency

Mental Model: The Spreadsheet as a Network of Information Pipelines



Imagine each cell as a processing station in a factory assembly line. Raw materials (input values) flow into some stations, get processed (formulas compute results), and then travel via conveyor belts to other stations that need them. The dependency graph is the blueprint of this factory—it maps every conveyor belt connection, showing exactly which stations supply which others. When a station changes its output (a cell value changes), we can trace the conveyor belts downstream to see exactly which other stations are affected and need to rerun their processing. Crucially, this blueprint also lets us detect when someone tries to create a circular conveyor belt that would cause materials to flow endlessly in a loop—a **circular reference** error.

The `DependencyGraph` is this blueprint. It maintains two core mappings: 1) from cell addresses to their `Cell` objects (the stations), and 2) from cell addresses to sets of other addresses they depend on (the conveyor belts). This dual representation—often called an adjacency list—enables efficient traversal in both directions: from dependents to dependencies (for evaluation order) and from dependencies to dependents (for change propagation).

DependencyGraph Data Structure Table

Field Name	Type	Description
`nodes`	`Map<string, Cell>`	Maps cell addresses (like `"A1"`) to their corresponding `Cell` objects. This provides O(1) access to any cell in the spreadsheet by its address. The graph doesn't own these cells—they're shared references to the same objects stored in the `Spreadsheet.grid`.
`edges`	`Map<string, Set<string>>`	Adjacency list representation. Keys are dependent cell addresses (cells that contain formulas). Values are Sets of dependency addresses (cells that are referenced in those formulas). For example, if cell `C1` contains `=A1+B1`, then `edges.get("C1")` would return `Set{"A1", "B1"}`.

Decision: Adjacency List vs. Adjacency Matrix for Dependency Graph

- **Context**: We need to represent directional relationships between cells (dependencies) and support efficient traversal for both topological sorting (dependencies before dependents) and change propagation (finding all dependents of a changed cell).
- **Options Considered**:
  1. **Adjacency Matrix**: 2D array where `matrix[i][j] = true` means cell i depends on cell j. Simple for small grids but O(n²) memory for `MAX_ROWS * MAX_COLUMNS` cells (26,000 entries for 1000×26).
  2. **Adjacency List**: Map from node to list of neighbors. Memory proportional to actual edges (formula references), which is typically sparse.
  3. **Implicit graph via cell.dependencies**: Store edges only in the `Cell` objects themselves, without a separate graph structure.
- **Decision**: Use adjacency list representation via `Map<string, Set<string>>`.
- **Rationale**: Spreadsheets are sparse—most cells are empty, and formulas typically reference only a handful of other cells. Adjacency list uses memory proportional to actual dependencies (edges), not potential cells (nodes). It also enables efficient forward traversal (from dependency to dependents) by building a reverse index when needed, which is crucial for change propagation.
- **Consequences**: We must maintain consistency between the `edges` map and the `dependencies` sets in individual `Cell` objects. Adding/removing a formula requires updating both places. The graph can be reconstructed from cell dependencies if needed.

Option	Pros	Cons	Chosen?
Adjacency Matrix	Simple queries, dense graphs	O(n²) memory, wasteful for sparse spreadsheets	❌
Adjacency List	**Memory efficient for sparse graphs, natural representation**	**Requires consistency maintenance, reverse traversal needs computation**	✅
Implicit (cell-only)	No duplication, simple	Hard to find dependents (requires scanning all cells)	❌

Graph Operation Examples

When cell `D4` changes from literal `5` to formula `=C3*2`:

1. **Add dependencies**: Parse formula `=C3*2` → find reference `"C3"` → `edges.set("D4", new Set(["C3"]))` and also `cellD4.dependencies.add("C3")`.
2. **Check for cycles**: Before adding edge `D4 → C3`, verify no path exists from `C3` back to `D4` (would create cycle).
3. **Propagate change**: When `C3` changes later, traverse `edges` to find `"D4"` in dependents of `"C3"`, mark `D4` as needing recalculation.

Topological Sort Algorithm for Recalculation

When a cell's value changes, we must recalculate all cells that depend on it (directly or indirectly) in the correct order: dependencies before dependents. This is achieved via **topological sort** on the dependency graph:

1. **Identify dirty cells**: Start from the changed cell, perform depth-first search (DFS) through the `edges` map to collect all transitively dependent cells.
2. **Build dependency counts**: For each dirty cell, count how many of its dependencies are also in the dirty set and not yet calculated.
3. **Process zero-dependency cells**: Initialize a queue with cells whose dependency count is zero (they depend only on already-calculated cells or the originally changed cell).
4. **Iterative reduction**: While queue not empty:
    1. Remove a cell from queue, recalculate its value.
    2. For each dependent of this cell (found via reverse edges), decrement its dependency count.
    3. If dependent's count reaches zero, add it to queue.
5. **Cycle detection**: If queue empties but some dirty cells remain unprocessed, a circular reference exists among them.

Circular Reference Detection Strategy

Circular references (A→B→C→A) must be detected and reported as errors rather than causing infinite recalculation loops:

1. **Preventive check**: Before adding a new dependency edge `X → Y`, perform depth-first search from `Y` to see if a path back to `X` already exists. If yes, reject the formula with a `#CIRCULAR!` error.
2. **Runtime detection**: During topological sort, if we cannot reduce all dirty cells' dependency counts to zero (step 5 above), identify the strongly connected component causing the cycle and mark all cells in it with `#CIRCULAR!` errors.
3. **User feedback**: Display `#CIRCULAR!` in all cells involved in the cycle, helping users debug the circular dependency chain.

Application State: Grid, Selection, and History

Mental Model: The Spreadsheet as a Time-Traveling Camera Over a Data Grid

Imagine a vast grid of data cells stretching beyond the visible screen. You're looking at this grid through a camera viewport that can pan around (scrolling). The camera has a spotlight highlighting one particular cell (selection). Most importantly, this setup includes a "time machine" recorder that captures snapshots of the entire grid before each change, allowing you to rewind or fast-forward through the history of edits. The `Spreadsheet` state object represents this entire setup: the data grid itself, the current camera position and spotlight, and the time machine's memory.

This comprehensive state object serves as the single source of truth for the entire application. All UI components read from it to render correctly, and all user actions ultimately modify it (through well-defined operations). The history mechanism layered on top enables the undo/redo functionality by storing incremental changes rather than full snapshots (to conserve memory).

Spreadsheet Data Structure Table

Field Name	Type	Description
`grid`	`Map<number, Map<number, Cell>>`	Two-level sparse array representation. Outer Map keys are row indices (0-based). Inner Map keys are column indices (0-based). Values are `Cell` objects. This sparse structure efficiently stores only cells that have data (values, formulas, or formatting), avoiding memory waste for empty cells.
`selection`	`{row: number, col: number}`	The currently selected cell's coordinates (0-based indices). The UI highlights this cell, and keyboard navigation moves this selection. Also determines which cell's formula appears in the formula bar.
`viewport`	`{topRow: number, leftCol: number, height: number, width: number}`	The visible portion of the grid in terms of row/column indices. Used by the virtualized grid rendering to know which cells to create DOM elements for. Typically initialized to show first `VIEWPORT_ROWS` rows and `VIEWPORT_COLUMNS` columns.
`history`	`Array<HistoryEntry>`	Stack of undoable operations. Each entry captures enough information to reverse (undo) and reapply (redo) a user action. We use a linear stack rather than tree for simplicity.
`historyIndex`	`number`	Current position in the undo/redo history. Points to the most recent applied state. When undo is performed, this decrements; when redo, it increments. Allows non-destructive undo (we don't discard redoable future states).

HistoryEntry Data Structure Table

Field Name	Type	Description
`action`	`string`	Type of operation: `"SET_VALUE"`, `"SET_FORMULA"`, `"SET_FORMAT"`, `"PASTE"`, etc. Determines how to apply the inverse operation during undo.
`cellAddress`	`string`	The cell address (e.g., `"B3"`) where the change occurred. For multi-cell operations like paste, this is the top-left cell of the affected range.
`previousState`	`any`	The previous value of whatever was changed. For `SET_VALUE`, this is the previous `value`; for `SET_FORMULA`, it's the previous `formula` string; for multi-cell operations, it's a mapping of addresses to their previous cell states.
`newState`	`any`	The new value after the change. Used for redo operations.

Decision: Sparse Nested Map vs. Dense 2D Array for Grid Storage

- **Context**: We need to store cells for a potentially large grid (1000×26 = 26,000 cells) where most cells are empty. Memory efficiency and access speed are both important.
- **Options Considered**:
  1. **Dense 2D array**: `Cell[][]` with `MAX_ROWS × MAX_COLUMNS` pre-allocated. O(1) access but wastes memory for empty cells (26,000 objects even if only 100 contain data).
  2. **Single flat Map**: `Map<string, Cell>` keyed by address strings like `"A1"`. Simple but requires string parsing for row/column operations.
  3. **Sparse nested Map**: `Map<number, Map<number, Cell>>` (row → column → cell). Efficient for sparse data, O(1) average access, natural row/column indexing.
- **Decision**: Use sparse nested Map structure.
- **Rationale**: Spreadsheets are typically sparse—users fill only a fraction of available cells. The nested Map uses memory proportional to actual data while maintaining fast O(1) access for both individual cells (`grid.get(row)?.get(col)`) and row-wise operations (get all cells in row 5: `grid.get(5)?.values()`). The 0-based numeric indices also align perfectly with the `ParsedCellReference` structure.
- **Consequences**: We must handle missing rows/columns gracefully (return `undefined` or default empty cell). Iteration over all cells requires nested loops but remains efficient because inner Maps exist only for non-empty rows.

Option	Pros	Cons	Chosen?
Dense 2D array	Fast access, simple iteration	Wastes memory for empty cells, large initial allocation	❌
Single flat Map	Simple, uniform access	Requires parsing addresses for row/col operations, unnatural range queries	❌
Sparse nested Map	**Memory efficient, natural 2D indexing, fast access**	**Slightly more complex iteration, null checks required**	✅

Grid Access Pattern Example

To get cell at address `"C5"` (column C = index 2, row 5 = index 4):

```javascript
const rowMap = spreadsheet.grid.get(4);  // Get row 5 (0-based)
if (rowMap) {
  const cell = rowMap.get(2);  // Get column C (index 2)
  return cell || createEmptyCell();
}
return createEmptyCell();
```

State Management Flow

When user edits cell `B3` to formula `=A1*2`:

1. **Capture previous state**: Create `HistoryEntry` with `action: "SET_FORMULA"`, `cellAddress: "B3"`, `previousState: oldFormula` (or `null` if previously empty), `newState: "=A1*2"`.
2. **Update grid**: Parse formula, update `Cell` object at `B3` with new formula and dependencies.
3. **Update dependency graph**: Remove old dependencies (if any), add new edge `B3 → A1`.

4. **Recalculate**: Trigger topological sort starting from `A1` (if it changed) or `B3` (if new formula).
5. **Push to history**: Add entry to `history` array, increment `historyIndex`.
6. **Update UI**: Re-render affected cells based on new values.

Common Pitfalls: Application State

⚠️ Pitfall: Storing full grid snapshots in undo history

- **Description**: Saving the entire `grid` Map for every undoable action.
- **Why it's wrong**: Quickly consumes huge amounts of memory (megabytes per snapshot for even moderately sized spreadsheets). Makes undo/redo operations slow due to deep copying.
- **Fix**: Store only incremental changes in `HistoryEntry`. For single-cell edits, store just the previous and new state of that cell. For multi-cell operations like paste, store only the affected cells' previous states.

⚠️ Pitfall: Mixing display state with data state

- **Description**: Storing UI-specific information like "isEditing" or "DOM element reference" in the core data model.
- **Why it's wrong**: Creates tight coupling between data and presentation, makes serialization difficult (can't serialize DOM references), and complicates state management.
- **Fix**: Keep the `Spreadsheet` state purely about data and selection. UI state (which cell is being edited, scroll position in pixels) belongs in the view layer components.

Implementation Guidance

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
State Management	Plain JavaScript objects with manual updates	Immutable.js or Immer for predictable state updates
Data Structures	Native ES6 Maps and Sets	Custom optimized structures for large grids
History Storage	In-memory array of operations	Persistent storage via IndexedDB or localStorage

B. Recommended File/Module Structure

```
spreadsheet-app/
├── src/
│   ├── data-model/           # Data structures and core logic
│   │   ├── cell.ts           # Cell, CellFormat, ParsedCellReference types
│   │   ├── spreadsheet.ts    # Spreadsheet state and grid operations
│   │   ├── dependency-graph.ts # DependencyGraph class
│   │   └── history.ts        # HistoryEntry and undo/redo logic
│   ├── parser/               # Formula parsing (separate component)
│   ├── ui/                   # Presentation layer
│   └── utils/
│       └── cell-address.ts   # Address parsing/formatting utilities
└── package.json
```

C. Infrastructure Starter Code

`src/utils/cell-address.ts` (Complete utility functions for cell address manipulation):

TYPESCRIPT

```typescript
// Constants
export const COLUMN_LETTERS = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ';
export const MAX_ROWS = 1000;
export const MAX_COLUMNS = 26;

// Error constants
export const ERRORS = {
  CIRCULAR_REF: '#CIRCULAR!',
  REF_ERROR: '#REF!',
  VALUE_ERROR: '#VALUE!',
  DIV_ZERO: '#DIV/0!',
  PARSE_ERROR: '#ERROR!'
};

// Convert 0-based column index to Excel column letters (A, B, ..., Z, AA, AB, ...)
export function columnToLetter(column: number): string {
  let letters = '';
  while (column >= 0) {
    letters = COLUMN_LETTERS[column % 26] + letters;
    column = Math.floor(column / 26) - 1;
  }
  return letters;
}

// Convert Excel column letters to 0-based column index
export function letterToColumn(letters: string): number {
  letters = letters.toUpperCase();
  let column = 0;
  for (let i = 0; i < letters.length; i++) {
    column = column * 26 + (letters.charCodeAt(i) - 65 + 1);
  }
  return column - 1; // Convert to 0-based
}
```

```
// Check if a string is a valid cell address (e.g., A1, $B$2, C$3, $D4)
export function isValidCellAddress(address: string): boolean {
  const match = address.match(/^\$?[A-Za-z]+\$?\d+$/);
  if (!match) return false;

  // Extract column and row parts
  const parts = address.match(/(\$?)([A-Za-z]+)(\$?)(\d+)/);
  if (!parts) return false;

  const [, colAbs, colLetters, rowAbs, rowNum] = parts;
  const column = letterToColumn(colLetters);
  const row = parseInt(rowNum, 10) - 1; // Convert to 0-based

  return row >= 0 && row < MAX_ROWS && column >= 0 && column < MAX_COLUMNS;
}

// Parse Excel-style cell address into structured format
export function parseCellAddress(address: string): ParsedCellReference {
  if (!isValidCellAddress(address)) {
    throw new Error(`Invalid cell address: ${address}`);
  }

  const parts = address.match(/(\$?)([A-Za-z]+)(\$?)(\d+)/)!;
  const [, colAbs, colLetters, rowAbs, rowNum] = parts;

  return {
    row: parseInt(rowNum, 10) - 1, // Convert to 0-based
    column: letterToColumn(colLetters),
    rowAbsolute: rowAbs === '$',
    columnAbsolute: colAbs === '$'
  };
}
```

```
// Convert row/column indices to cell address string

export function toCellAddress(

  row: number,

  column: number,

  rowAbsolute: boolean = false,

  columnAbsolute: boolean = false

): string {

  if (row < 0 || row >= MAX_ROWS || column < 0 || column >= MAX_COLUMNS) {

    throw new Error(`Cell coordinates out of bounds: row ${row}, column ${column}`);

  }



  const rowPart = `${rowAbsolute ? '$' : ''}${row + 1}`;

  const colPart = `${columnAbsolute ? '$' : ''}${columnToLetter(column)}`;

  return colPart + rowPart;

}
```

D. Core Logic Skeleton Code

`src/data-model/cell.ts` (Core data type definitions):

TYPESCRIPT

```typescript
// Types for cell formatting
export interface CellFormat {
  numberFormat: string;      // e.g., "currency", "percentage", "0.00"
  decimalPlaces: number | null;
  alignment: 'left' | 'center' | 'right';
  bold: boolean;
  italic: boolean;
}

// Parsed cell reference from formula
export interface ParsedCellReference {
  row: number;           // 0-based row index
  column: number;        // 0-based column index
  rowAbsolute: boolean;  // Whether row is absolute ($)
  columnAbsolute: boolean; // Whether column is absolute ($)
}

// Main cell data structure
export interface Cell {
  value: any;                   // Current computed/stored value
  formula: string;              // Raw formula string (empty if not a formula)
  format: CellFormat;           // Visual formatting
  dependencies: Set<string>;    // Cell addresses this formula references
}

// Factory functions for creating cells
export function createEmptyCell(): Cell {
  return {
    value: '',
    formula: '',
    format: {
      numberFormat: 'general',
      decimalPlaces: null,
      alignment: 'left',
```

```
      bold: false,
      italic: false
    },
    dependencies: new Set<string>()
  };
}

export function createNumberCell(value: number): Cell {
  const cell = createEmptyCell();
  cell.value = value;
  return cell;
}

export function createFormulaCell(formula: string): Cell {
  const cell = createEmptyCell();
  cell.formula = formula;
  // Note: value will be computed later when formula is evaluated
  // dependencies will be populated when formula is parsed
  return cell;
}
```

`src/data-model/spreadsheet.ts` (Spreadsheet state management):

TYPESCRIPT

```typescript
import { Cell, createEmptyCell } from './cell';
import { parseCellAddress, toCellAddress, isValidCellAddress } from '../utils/cell-address';

// Main spreadsheet state
export interface Spreadsheet {
  grid: Map<number, Map<number, Cell>>;  // Sparse nested map: row -> column -> cell
  selection: { row: number; col: number };
  viewport: {
    topRow: number;
    leftCol: number;
    height: number;
    width: number;
  };
  history: Array<any>;  // Will be typed properly in history.ts
  historyIndex: number;
}

// Initialize a new empty spreadsheet
export function createSpreadsheet(): Spreadsheet {
  return {
    grid: new Map(),
    selection: { row: 0, col: 0 },
    viewport: {
      topRow: 0,
      leftCol: 0,
      height: 20,  // VIEWPORT_ROWS
      width: 10    // VIEWPORT_COLUMNS
    },
    history: [],
    historyIndex: -1
  };
}

// Get cell at specified row and column (0-based indices)
```

```typescript
export function getCell(spreadsheet: Spreadsheet, row: number, col: number): Cell {
  // TODO 1: Get the row map from spreadsheet.grid using row as key
  // TODO 2: If row map exists, get the cell from it using col as key
  // TODO 3: If cell exists, return it; otherwise return a new empty cell
  // Hint: Use createEmptyCell() for default cells
}

// Set raw value for a cell (not a formula)
export function setCellValue(spreadsheet: Spreadsheet, row: number, col: number, value: any): void {
  // TODO 1: Get or create the row map in spreadsheet.grid
  // TODO 2: Get or create the cell at [row][col]
  // TODO 3: Update cell.value with the new value
  // TODO 4: Clear cell.formula (since we're setting a raw value, not a formula)
  // TODO 5: Clear cell.dependencies (raw values don't have dependencies)
  // TODO 6: Trigger dependency graph update to remove old dependencies
}

// Set formula for a cell
export function setCellFormula(spreadsheet: Spreadsheet, row: number, col: number, formula: string): void {
  // TODO 1: Get or create the row map in spreadsheet.grid
  // TODO 2: Get or create the cell at [row][col]
  // TODO 3: Update cell.formula with the new formula string
  // TODO 4: Parse the formula to extract cell references (will be implemented in parser component)
  // TODO 5: Update cell.dependencies with the parsed references
  // TODO 6: Evaluate the formula to compute cell.value (will use dependency graph)
  // TODO 7: Update dependency graph with new edges
}

// Get cell by address string (e.g., 'A1')
export function getCellByAddress(spreadsheet: Spreadsheet, address: string): Cell {
  // TODO 1: Parse the address using parseCellAddress()
  // TODO 2: Use getCell() with the parsed row and column
  // TODO 3: Return the cell
```

```
}


// Get all cells that have data (non-empty iterator for efficient traversal)
export function* getAllCells(spreadsheet: Spreadsheet): Generator<[number, number, Cell]> {
  // TODO 1: Iterate over all rows in spreadsheet.grid (outer Map)
  // TODO 2: For each row, iterate over all columns in the row map (inner Map)
  // TODO 3: Yield [row, col, cell] for each non-empty cell
  // Hint: Use for...of loops with .entries() on Maps
}
```

`src/data-model/dependency-graph.ts` (Dependency graph operations):

TYPESCRIPT

```typescript
import { Cell } from './cell';

export class DependencyGraph {
  nodes: Map<string, Cell> = new Map();        // address -> Cell (shared references)
  edges: Map<string, Set<string>> = new Map(); // dependent -> Set<dependency>

  // Add a dependency relationship: dependentCell depends on dependencyCell
  addDependency(dependentAddress: string, dependencyAddress: string): void {
    // TODO 1: Check if adding this edge would create a circular reference
    // TODO 2: If circular reference detected, throw error with ERRORS.CIRCULAR_REF
    // TODO 3: Get or create the edge set for dependentAddress
    // TODO 4: Add dependencyAddress to the edge set
    // TODO 5: Also update the Cell's dependencies set (in the node)
    // TODO 6: Ensure reverse indexing for efficient dependent lookup
  }

  // Remove a dependency relationship
  removeDependency(dependentAddress: string, dependencyAddress: string): void {
    // TODO 1: Get the edge set for dependentAddress
    // TODO 2: If set exists, remove dependencyAddress from it
    // TODO 3: If edge set becomes empty, remove the entry from edges map
    // TODO 4: Also update the Cell's dependencies set (in the node)
    // TODO 5: Update reverse indexing
  }

  // Get all cells that directly depend on the given cell
  getDependents(cellAddress: string): Set<string> {
    // TODO 1: Initialize an empty Set for results
    // TODO 2: Iterate through all entries in edges map
    // TODO 3: For each [dependent, dependencies] pair, check if dependencies includes cellAddress
    // TODO 4: If yes, add dependent to results set
    // TODO 5: Return results
  }
```

```
  // Get topological order for recalculation (dependencies before dependents)
  getTopologicalOrder(changedCell: string): string[] {
    // TODO 1: Perform DFS from changedCell to collect all transitive dependents
    // TODO 2: Build a map of dependency counts for each cell in the affected set
    // TODO 3: Initialize queue with cells that have zero dependencies in the affected set
    // TODO 4: While queue is not empty:
    //   a. Remove cell from queue, add to result order
    //   b. For each dependent of this cell, decrement its dependency count
    //   c. If dependent's count reaches zero, add it to queue
    // TODO 5: If result order length < affected set size, circular reference detected
    // TODO 6: Return the topological order
  }

  // Detect if adding an edge would create a circular reference
  detectCircularReference(startAddress: string, targetAddress: string): boolean {
    // TODO 1: If startAddress equals targetAddress, return true (self-reference)
    // TODO 2: Perform DFS from targetAddress, following edges outward
    // TODO 3: If we encounter startAddress during DFS, circular reference exists
    // TODO 4: Use visited Set to avoid infinite loops in already-cyclic graphs
    // TODO 5: Return true if circular, false otherwise
  }
}
```

E. Language-Specific Hints

1. **Maps and Sets**: Use ES6 `Map` and `Set` for their key ordering (insertion order) and O(1) operations. Remember that object keys in regular JavaScript objects are always strings, which is why we need `Map` for numeric row indices.
2. **Sparse Arrays**: The nested Map structure (`Map<number, Map<number, Cell>>`) effectively creates a sparse 2D array. Access patterns should always check for existence: `const row = grid.get(rowIndex); if (row) { const cell = row.get(colIndex); }` .
3. **Default Cells**: When `getCell()` is called for an empty cell, return a default empty cell rather than `undefined` . This simplifies calling code but means we need factory functions like `createEmptyCell()` .
4. **Shared References**: The `DependencyGraph.nodes` Map contains references to the same `Cell` objects stored in the `Spreadsheet.grid` . This is intentional—it avoids duplication and ensures consistency. Be careful not to mutate one without updating the other.

5. **Error Objects**: Use the `ERRORS` constants for spreadsheet error values like `#CIRCULAR!`. These are special values that propagate through formulas and display in cells.

F. Milestone Checkpoint

After implementing the data model (end of Milestone 1 foundation):

1. **Run the type checker**: If using TypeScript, run `tsc --noEmit` to verify all types are correctly defined and used.
2. **Unit test the address utilities**:

```bash
# Create a simple test file and run with Node
node test-cell-address.js
```

Expected output: All tests pass for `columnToLetter`, `letterToColumn`, `parseCellAddress`, `toCellAddress`, and `isValidCellAddress`.

3. **Verify sparse grid operations**:
   - Create a spreadsheet with `createSpreadsheet()`
   - Set values at A1, B2, C3 using `setCellValue()`
   - Verify `getCellByAddress()` returns correct values
   - Verify `getAllCells()` yields exactly 3 cells (not 26,000)
   - Check that empty cells return default empty cells via `getCell()`
4. **Signs of problems**:
   - **"Cannot read property 'get' of undefined"**: You're accessing a row that doesn't exist in the grid Map. Use `grid.get(row)?.get(col)` or check existence first.
   - **Address parsing fails for columns beyond Z**: The `columnToLetter` and `letterToColumn` functions must handle multiple letters (AA, AB, etc.).
   - **Memory usage grows rapidly**: You might be storing full cell objects for empty cells instead of using sparse Maps.

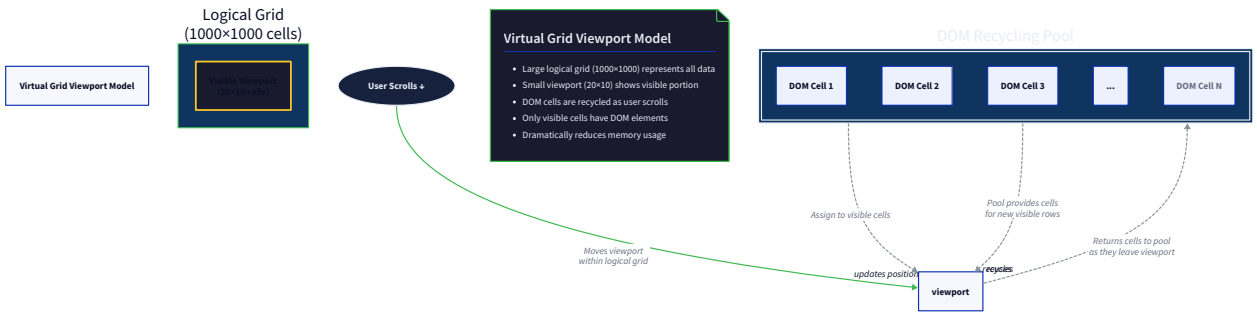
Component Design: Virtualized Grid

> **Milestone(s):** Milestone 1 (Grid & Cell Rendering)

The virtualized grid component is the visual interface through which users interact with the spreadsheet. Unlike simplistic implementations that render thousands of DOM elements simultaneously, this component must manage a large logical grid (1,000×26 cells) while maintaining responsive performance. The core challenge is providing the illusion of a complete grid while only creating and managing DOM elements for the visible portion, efficiently updating them as the user scrolls.

Mental Model: A Movable Viewport Over a Large Canvas

Imagine the spreadsheet as a massive canvas painted with a grid of 1,000 rows and 26 columns. The user's screen is a small, movable window (the **viewport**) that shows only a tiny portion of this canvas—perhaps 20 rows and 10 columns. As the user scrolls, the window moves across the canvas, revealing different sections.



Instead of painting the entire canvas (which would be impossibly large and slow), we paint only the visible area on the window glass. When the window moves, we don't repaint the entire canvas—we simply **reposition existing paint** (DOM elements) and fill in the newly revealed areas. This technique, called **virtual scrolling**, is like having a fixed set of movable sticky notes that we constantly reposition and relabel as the user's view changes.

The critical insight is that **the DOM is expensive, but moving existing DOM elements is cheap**. We maintain a small, fixed pool of cell elements (exactly `VIEWPORT_ROWS` × `VIEWPORT_COLUMNS` elements) that we reuse for different logical positions. When the user scrolls, we update the cell content and CSS positioning of these elements to match the new visible region, creating a seamless scrolling experience.

Grid Interface and Methods

The virtual grid component provides a clean interface that separates grid rendering logic from spreadsheet business logic. It receives data (cell values) and state (selection) from the data model and translates them into visual representations.

Table: Virtual Grid Component Interface

Method Name	Parameters	Returns	Description
`initializeGrid`	`containerElement: HTMLElement, spreadsheet: Spreadsheet`	`void`	Mounts the grid into the specified container DOM element and establishes bidirectional communication with the spreadsheet data model
`renderViewport`	`viewport: {topRow: number, leftCol: number, height: number, width: number}`	`void`	Updates all visible cells to reflect the current viewport position. Called on scroll, resize, or data change
`getCellElement`	`logicalRow: number, logicalCol: number`	`HTMLElement or null`	Returns the DOM element for a specific logical cell if it's currently in the viewport, otherwise returns null
`setSelection`	`row: number, col: number`	`void`	Visually highlights the specified cell as selected, clearing previous selection
`enterEditMode`	`row: number, col: number, initialValue: string`	`void`	Replaces the cell display with an editable input field positioned at the specified cell
`exitEditMode`	`saveValue: boolean`	`void`	Exits edit mode, either saving the edited value back to the data model or discarding changes
`scrollToCell`	`row: number, col: number`	`void`	Adjusts the viewport to make the specified cell visible, scrolling if necessary
`refreshCell`	`row: number, col: number`	`void`	Updates the visual representation of a single cell without re-rendering the entire viewport

Table: Grid Data Structure (Internal State)

Field Name	Type	Description
`containerElement`	`HTMLElement`	The root DOM element that contains the entire grid assembly
`viewportElement`	`HTMLElement`	The scrollable container that creates the virtual scrolling effect
`cellPool`	`Array<Array<HTMLElement>>`	2D array of DOM elements (size: `VIEWPORT_ROWS` × `VIEWPORT_COLUMNS`) that are recycled for different logical positions
`headers`	`{rowHeaders: HTMLElement[], columnHeaders: HTMLElement[]}`	DOM elements for row numbers (1, 2, 3...) and column letters (A, B, C...)
`editInput`	`HTMLInputElement or null`	The single input element used for cell editing (created once, repositioned as needed)
`currentViewport`	`{topRow: number, leftCol: number, height: number, width: number}`	Tracks the currently visible logical region of the grid
`selection`	`{row: number, col: number} or null`	The currently selected cell in logical coordinates
`spreadsheetRef`	`Spreadsheet`	Reference to the spreadsheet data model for retrieving cell values and formats

Virtual Scrolling Algorithm

The virtual scrolling algorithm manages the relationship between logical cell positions (in the full spreadsheet) and physical DOM elements (in the viewport). It ensures that when the user scrolls, the correct cells are displayed with minimal DOM manipulation.

Algorithm Steps:

1. **Initial Setup:**
    1. Create a fixed grid of DOM elements (`VIEWPORT_ROWS` × `VIEWPORT_COLUMNS` cells) with absolute positioning
    2. Create row and column header elements (26 column headers, `VIEWPORT_ROWS` row headers)
    3. Set up a single hidden input element for cell editing
    4. Attach scroll event listeners to the viewport container
2. **On Viewport Scroll:**
    1. Calculate the new logical viewport bounds:
        - `topRow = Math.floor(viewportElement.scrollTop / CELL_HEIGHT)`
        - `leftCol = Math.floor(viewportElement.scrollLeft / CELL_WIDTH)`
    2. If the viewport hasn't changed significantly (within a small threshold), do nothing (performance optimization)
    3. Determine which logical cells are now visible in the viewport rectangle
    4. For each cell in the DOM pool:
        - Calculate its target logical position: `logicalRow = topRow + poolRowIndex`, `logicalCol = leftCol + poolColIndex`
        - If the logical position is within spreadsheet bounds (`0 ≤ logicalRow < MAX_ROWS`, `0 ≤ logicalCol < MAX_COLUMNS`):
            - Update the cell's CSS `top` and `left` properties: `(poolRowIndex * CELL_HEIGHT)px`, `(poolColIndex * CELL_WIDTH)px`

        - Retrieve cell data from the spreadsheet: `cell = spreadsheet.getCell(logicalRow, logicalCol)`
        - Update the cell's content (value or formula result) and formatting
        - Update the cell's `data-row` and `data-col` attributes for event handling
    - Else (cell is outside bounds):
        - Hide the cell element (set `display: none`)

3. **On Data Change:**
    1. When a cell value changes in the spreadsheet data model:
    2. If the changed cell is currently visible in the viewport:
        - Find the corresponding DOM element in the cell pool
        - Update its displayed content
    3. If the change affects selection or editing state, update those visual states separately

4. **On Resize:**
    1. Recalculate viewport dimensions based on container size
    2. Adjust the scrollable area's dimensions: `height = MAX_ROWS * CELL_HEIGHT`, `width = MAX_COLUMNS * CELL_WIDTH`
    3. Re-render the viewport at the current scroll position

> **Key Insight:** The DOM pool elements always remain children of the viewport container at fixed positions relative to the container. What changes are: (1) their CSS positioning (to stay visually in the same screen position as we scroll), and (2) their content (to show different logical cells). The scrollbars control which logical region maps to the fixed screen region.

Table: Coordinate Transformation Example

Scenario	Scroll Position	Logical Top Row	Physical Top Row (in pool)	Logical Cell A1 Position
Initial	0px vertical, 0px horizontal	0	Pool index 0	Displayed in pool cell [0,0]
Scrolled down 200px	200px vertical, 0px horizontal	10 (if CELL_HEIGHT=20)	Pool index 0	Not visible (above viewport)
Same scroll	200px vertical, 0px horizontal	10	Pool index 0	Logical cell [10,0] displayed in pool cell [0,0]

Architecture Decision Records for Grid

Decision: Virtual Scrolling with DOM Recycling

- **Context**: The spreadsheet must display up to 26,000 cells (1,000 rows × 26 columns) but typical screens can only show about 200 cells (20 rows × 10 columns) at once. Rendering all cells would create 26,000 DOM elements, causing severe performance issues in memory, layout calculation, and rendering.
- **Options Considered**:
  1. **Render All Cells**: Create DOM elements for every cell in the spreadsheet
  2. **Virtual Scrolling with DOM Recycling**: Maintain a fixed pool of DOM elements for the visible viewport, updating content and position as the user scrolls
  3. **Canvas/WebGL Rendering**: Draw the entire grid on a single canvas element
- **Decision**: Virtual scrolling with DOM recycling (Option 2)
- **Rationale**:
  - Option 1 fails with large grids (tested: 26,000 divs causes ~2 second render time and 300MB memory)
  - Option 3 provides best performance for extremely large grids but sacrifices accessibility (screen readers), text selection, native form inputs for editing, and CSS styling flexibility
  - Option 2 offers near-native scrolling performance while preserving full DOM capabilities for editing, styling, and accessibility
  - Modern spreadsheet applications (Google Sheets, Excel Online) use similar DOM recycling techniques
- **Consequences**:
  - We must implement scroll position synchronization logic
  - Cell event handlers must be delegated (attached to container, not individual cells)
  - Selection highlighting requires careful coordination between logical and physical coordinates

Table: Grid Rendering Options Comparison

Option	Pros	Cons	Chosen?
Render All Cells	Simple implementation, no scroll sync logic	Unusable performance with 26,000+ cells, memory explosion	No
Virtual Scrolling with DOM Recycling	Excellent performance, preserves DOM capabilities, accessible	Complex implementation, requires scroll synchronization	**Yes**
Canvas/WebGL Rendering	Ultimate performance for huge grids, smooth animations	No native text selection/editing, poor accessibility, complex hit testing	No

Decision: Single Edit Input Element

- **Context**: When editing a cell, we need a text input field positioned exactly over the cell. Multiple approaches exist for managing this input element.
- **Options Considered**:
  1. **One Input Per Cell**: Each cell DOM element contains its own `<input>` (hidden when not editing)
  2. **Single Shared Input**: One input element that's moved and repositioned when entering edit mode
  3. **ContentEditable**: Use `contenteditable` attribute on cell elements instead of separate input
- **Decision**: Single shared input element (Option 2)
- **Rationale**:
  - Option 1 adds significant DOM overhead (hundreds of hidden inputs) and complicates focus management
  - Option 3 provides less control over editing behavior and validation
  - Option 2 minimizes DOM nodes, provides full control over input behavior, and matches professional spreadsheet implementations
  - The input can be absolutely positioned with CSS transforms for pixel-perfect alignment
- **Consequences**:
  - Must carefully manage input visibility and positioning
  - Need to save/cancel edit state when moving between cells
  - Keyboard navigation must transfer focus to the shared input

Decision: Sparse Data Model Integration

- **Context**: The grid needs to display cell values, but most cells are empty. We need an efficient way to retrieve cell data.
- **Options Considered**:
  1. **Dense Array**: 2D array of 26,000 cell objects (most with default values)
  2. **Sparse Map**: Map data structure storing only non-empty cells
  3. **Hybrid**: Row-based sparse storage (Map of row Maps)
- **Decision**: Hybrid sparse storage (`Spreadsheet.grid` as `Map<number, Map<number, Cell>>`)
- **Rationale**:
  - Option 1 wastes memory on empty cells (approximately 26MB for 26,000 objects)
  - Option 2 (single Map with composite keys) is efficient but complicates range queries
  - Option 3 provides efficient memory use while enabling row-based operations and range iteration
  - The `Spreadsheet.getCell(row, col)` method can return a default empty cell when none exists
- **Consequences**:
  - Grid rendering must handle missing cell data gracefully
  - Cell lookup has O(1) average complexity but requires two Map lookups

Common Pitfalls: Grid Rendering

⚠️ Pitfall: Incorrect Scrollbar Dimensions

- **Description**: Setting the scrollable area dimensions to the viewport size instead of the full grid size, resulting in no scrollbars appearing
- **Why It's Wrong**: The browser shows scrollbars based on the difference between container size and content size. If the content is the same size as the container, no scrolling occurs

- **Fix**: Ensure the scrollable content element has CSS dimensions equal to the full logical grid: `width = MAX_COLUMNS * CELL_WIDTH`, `height = MAX_ROWS * CELL_HEIGHT`

⚠️ **Pitfall: Event Handler Per Cell**

- **Description**: Attaching click, double-click, or focus event listeners to each individual cell DOM element
- **Why It's Wrong**: As cells are recycled during scrolling, event handlers would need to be constantly detached and reattached, causing memory leaks and incorrect behavior
- **Fix**: Use event delegation—attach a single event listener to the grid container and use event bubbling. Check the event target's `data-row` and `data-col` attributes to determine which logical cell was clicked

⚠️ **Pitfall: Synchronous Layout Thrashing**

- **Description**: Reading DOM properties (like `offsetWidth`) then immediately writing to the DOM (like setting `style.top`), causing the browser to recalculate layout multiple times
- **Why It's Wrong**: Modern browsers batch DOM reads and writes separately. Interleaving reads and writes forces immediate layout calculations, killing scrolling performance
- **Fix**: Batch all DOM reads first, then all DOM writes. Use `requestAnimationFrame` for visual updates to align with the browser's paint cycle

⚠️ **Pitfall: Forgetting to Update Headers**

- **Description**: When scrolling horizontally, column headers (A, B, C...) don't scroll with the content, making it impossible to know which column you're viewing
- **Why It's Wrong**: Headers provide critical navigation context. Static headers disconnect from the data being viewed
- **Fix**: Implement header synchronization—update the displayed column letters based on `leftCol` viewport position, and row numbers based on `topRow`

⚠️ **Pitfall: Selection Visuals on Scrolled Cells**

- **Description**: The selected cell highlight appears on the wrong visual cell after scrolling because selection coordinates aren't transformed between logical and viewport space
- **Why It's Wrong**: Selection is stored in logical coordinates (row=5, col=3) but must be rendered in viewport-relative coordinates. Without transformation, the highlight stays at a fixed screen position
- **Fix**: When rendering the viewport, check each cell's logical coordinates against the selection coordinates and apply highlighting CSS only when they match

Implementation Guidance: Grid Component

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
Grid Rendering	Fixed-size DIV elements with absolute positioning	CSS Grid layout with transform-based positioning
Virtual Scrolling	Manual scroll event handling with `scrollTop` / `scrollLeft`	Intersection Observer API for predictive rendering
Cell Editing	Single shared `HTMLInputElement`	ContentEditable with custom caret positioning
Performance Optimization	Basic requestAnimationFrame batching	Virtualized cell measurement with resize observer

B. Recommended File/Module Structure

```
spreadsheet-app/
├── src/
│   ├── components/
│   │   ├── VirtualGrid/
│   │   │   ├── VirtualGrid.js          # Main grid component
│   │   │   ├── VirtualGrid.css         # Grid styling
│   │   │   ├── CellRenderer.js         # Cell content rendering
│   │   │   └── ScrollManager.js        # Virtual scroll logic
│   │   ├── FormulaEditor/
│   │   └── Toolbar/
│   ├── core/
│   │   ├── Spreadsheet.js             # Data model (from Data Model section)
│   │   ├── Parser.js                  # Formula parser
│   │   └── DependencyGraph.js         # Dependency tracking
│   └── utils/
│       ├── cellAddress.js             # toCellAddress, parseCellAddress, etc.
│       └── domHelpers.js              # DOM manipulation utilities
```

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

JAVASCRIPT

```javascript
// src/utils/domHelpers.js - Complete DOM utilities

export function createElement(tag, className, attributes = {}) {

    const element = document.createElement(tag);

    if (className) element.className = className;

    Object.keys(attributes).forEach(key => {

        element.setAttribute(key, attributes[key]);

    });

    return element;

}


export function removeAllChildren(element) {

    while (element.firstChild) {

        element.removeChild(element.firstChild);

    }

}


export function debounce(func, wait) {

    let timeout;

    return function executedFunction(...args) {

        const later = () => {

            clearTimeout(timeout);

            func(...args);

        };

        clearTimeout(timeout);

        timeout = setTimeout(later, wait);

    };

}


// src/utils/cellAddress.js - Complete cell address utilities

export function columnToLetter(column) {

    let letters = '';

    while (column >= 0) {

        letters = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'[column % 26] + letters;

        column = Math.floor(column / 26) - 1;
```

```
    }

    return letters;

}

export function letterToColumn(letters) {

    let column = 0;

    for (let i = 0; i < letters.length; i++) {

        column *= 26;

        column += letters.charCodeAt(i) - 'A'.charCodeAt(0) + 1;

    }

    return column - 1; // Convert to 0-based

}

export function parseCellAddress(address) {

    // Implementation from Data Model section

    const match = address.match(/^(\$?)([A-Z]+)(\$?)(\d+)$/);

    if (!match) return null;



    const [, colAbs, colLetters, rowAbs, rowNum] = match;

    return {

        row: parseInt(rowNum) - 1, // Convert to 0-based

        column: letterToColumn(colLetters),

        rowAbsolute: rowAbs === '$',

        columnAbsolute: colAbs === '$'

    };

}

export function toCellAddress(row, column, rowAbsolute = false, columnAbsolute = false) {

    const rowPart = `${rowAbsolute ? '$' : ''}${row + 1}`;

    const colPart = `${columnAbsolute ? '$' : ''}${columnToLetter(column)}`;

    return colPart + rowPart;

}

export function isValidCellAddress(address) {
```

```
    return parseCellAddress(address) !== null;
}
```

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

JAVASCRIPT

```javascript
// src/components/VirtualGrid/VirtualGrid.js

import { createElement, debounce } from '../../utils/domHelpers.js';

import { columnToLetter } from '../../utils/cellAddress.js';


export class VirtualGrid {

    constructor(containerElement, spreadsheet) {

        this.container = containerElement;

        this.spreadsheet = spreadsheet;

        this.cellPool = [];

        this.currentViewport = { topRow: 0, leftCol: 0, height: 0, width: 0 };

        this.selection = null;

        this.editInput = null;



        // Constants

        this.CELL_HEIGHT = 25;

        this.CELL_WIDTH = 100;

        this.VIEWPORT_ROWS = 20;

        this.VIEWPORT_COLUMNS = 10;



        this.initializeGrid();

    }



    initializeGrid() {

        // TODO 1: Clear container and create grid structure

        // TODO 2: Create viewport container with overflow: auto

        // TODO 3: Create header elements (row numbers and column letters)

        // TODO 4: Create cell pool (VIEWPORT_ROWS × VIEWPORT_COLUMNS DIV elements)

        // TODO 5: Position cell pool elements absolutely within viewport

        // TODO 6: Create single edit input (initially hidden)

        // TODO 7: Set up scroll event listener with debouncing

        // TODO 8: Set up click/double-click event delegation

        // TODO 9: Set initial scrollable area dimensions: MAX_ROWS * CELL_HEIGHT, MAX_COLUMNS * CELL_WIDTH

        // TODO 10: Trigger initial renderViewport call
```

```
    }

    renderViewport() {
        // TODO 1: Calculate new viewport bounds from scroll positions
        // TODO 2: If viewport hasn't changed significantly, return early (performance)
        // TODO 3: Update column headers to show letters for visible columns (leftCol to leftCol + VIEWPORT_COLUMNS)
        // TODO 4: Update row headers to show numbers for visible rows (topRow to topRow + VIEWPORT_ROWS)
        // TODO 5: For each cell in the pool:
        //   TODO 5a: Calculate logical position: logicalRow = topRow + poolRow, logicalCol = leftCol + poolCol
        //   TODO 5b: If logical position is within bounds (0 ≤ row < MAX_ROWS, 0 ≤ col < MAX_COLUMNS):
        //     TODO 5b-i: Update cell's CSS top/left: (poolRow * CELL_HEIGHT)px, (poolCol * CELL_WIDTH)px
        //     TODO 5b-ii: Get cell data: cell = this.spreadsheet.getCell(logicalRow, logicalCol)
        //     TODO 5b-iii: Update cell content: display cell.value or formula result
        //     TODO 5b-iv: Apply formatting (bold, italic, alignment from cell.format)
        //     TODO 5b-v: Set data attributes: data-row=logicalRow, data-col=logicalCol
        //     TODO 5b-vi: Apply selection highlight if (logicalRow, logicalCol) matches this.selection
        //     TODO 5b-vii: Show cell (display: block)
        //   TODO 5c: Else (outside bounds):
        //     TODO 5c-i: Hide cell (display: none)
        // TODO 6: Update this.currentViewport with new bounds
    }

    setSelection(row, col) {
        // TODO 1: Clear previous selection highlight from all cells
        // TODO 2: Update this.selection = {row, col}
        // TODO 3: If selected cell is in current viewport:
        //   TODO 3a: Find corresponding pool cell
        //   TODO 3b: Apply selection CSS class
        // TODO 4: If selected cell is NOT in current viewport:
        //   TODO 4a: Call scrollToCell(row, col) to make it visible
    }
```

```
    enterEditMode(row, col, initialValue = '') {
        // TODO 1: Exit any active edit mode (save or cancel as appropriate)
        // TODO 2: Find the DOM element for this cell in the pool
        // TODO 3: Position the shared editInput over the cell:
        //   TODO 3a: Calculate pixel position: left = (col - leftCol) * CELL_WIDTH, top = (row - topRow) * CELL_HEIGHT
        //   TODO 3b: Set editInput.style.transform = `translate(${left}px, ${top}px)`
        //   TODO 3c: Set editInput.style.width = `${CELL_WIDTH}px`
        //   TODO 3d: Set editInput.style.height = `${CELL_HEIGHT}px`
        // TODO 4: Set editInput.value = initialValue
        // TODO 5: Show editInput (display: block)
        // TODO 6: Focus editInput and select all text
        // TODO 7: Hide the cell's normal content (set opacity: 0 on cell element)
        // TODO 8: Add event listeners for editInput (Enter to save, Escape to cancel, Tab to move)
    }

    scrollToCell(row, col) {
        // TODO 1: Calculate required scroll positions:
        //   scrollTop = row * CELL_HEIGHT
        //   scrollLeft = col * CELL_WIDTH
        // TODO 2: Adjust scroll positions to keep cell centered in viewport if possible
        // TODO 3: Smoothly scroll viewport container to new positions
        // TODO 4: After scroll completes (use scroll event or setTimeout), renderViewport()
    }
}

// src/components/VirtualGrid/ScrollManager.js
export class ScrollManager {
    constructor(viewportElement, onScroll) {
        this.viewport = viewportElement;
        this.onScroll = debounce(onScroll, 16); // ~60fps
```

```
        // TODO 1: Add scroll event listener to viewport
        // TODO 2: Implement scroll position caching to avoid unnecessary renders
        // TODO 3: Implement predictive rendering (pre-render rows/columns about to enter viewport)
    }
}
```

E. Language-Specific Hints

1. **Use CSS Transforms for Positioning**: Instead of setting `style.top` and `style.left`, use `style.transform = translate3d(x, y, 0)` for hardware acceleration.
2. **Batch DOM Reads/Writes**: Use `requestAnimationFrame` to batch visual updates and avoid layout thrashing.
3. **Event Delegation Pattern**: Attach a single event listener to the grid container, not individual cells. Use `event.target.closest('[data-row]')` to find the clicked cell.
4. **CSS Containment**: Apply `contain: strict` to cell elements to isolate their rendering and improve performance.
5. **ResizeObserver**: Use `ResizeObserver` to handle window resizing efficiently instead of listening to the window `resize` event.

F. Milestone Checkpoint

After implementing the virtual grid component, verify the following:

1. **Rendering Test**: Open the spreadsheet. You should see a 20×10 grid of cells (200 total DIV elements), not 26,000.
2. **Scroll Test**: Scroll vertically and horizontally. New cells should appear smoothly without flickering or gaps. Scrollbars should reflect the full grid size (1,000 rows, 26 columns).
3. **Header Sync Test**: When scrolling horizontally, column headers should update (showing letters for visible columns). When scrolling vertically, row numbers should update.
4. **Selection Test**: Click a cell—it should highlight with a distinct border or background color. Scroll away and back—the selection should persist on the correct logical cell.
5. **Edit Test**: Double-click a cell. An input field should appear over the cell with the cell's current value. Press Enter to save, Escape to cancel.
6. **Performance Test**: Open browser DevTools Performance panel, record scrolling. There should be no jank (< 16ms per frame). Memory usage should be stable (< 50MB even with scrolling).

G. Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
No scrollbars appear	Scrollable content same size as viewport	Check CSS: viewport height/width vs content height/width	Ensure content element has dimensions = MAX_ROWS * CELL_HEIGHT, MAX_COLUMNS * CELL_WIDTH
Cells jump/disappear while scrolling	Incorrect pool cell positioning	Log pool indices vs logical positions during render	Verify: logicalRow = topRow + poolRowIndex, logicalCol = leftCol + poolColIndex
Selection highlights wrong cell after scroll	Selection coordinates not transformed	Check if selection CSS is applied to pool index instead of logical position	Compare cell's data-row/data-col attributes with this.selection before applying highlight
Edit input appears in wrong place	Incorrect position calculation	Console.log editInput transform vs cell position	Ensure: left = (col - leftCol) * CELL_WIDTH, top = (row - topRow) * CELL_HEIGHT
Memory grows with scrolling	Event listeners not cleaned up	Use Chrome DevTools Memory profiler	Use event delegation (one listener on container) instead of per-cell listeners
Scroll performance janky	Too many reflows/repaints	Chrome DevTools Performance panel shows long paint times	Batch DOM updates with requestAnimationFrame, use CSS transforms instead of top/left

---

Component Design: Formula Parser and Evaluator

> **Milestone(s):** Milestone 2 (Formula Parser) and Milestone 3 (Dependency Graph & Recalculation) — The parser component converts raw formula strings into evaluable expressions, which drives both formula evaluation and dependency graph construction.

Mental Model: A Calculator That Understands Spreadsheet Addresses

Imagine a regular calculator that can perform arithmetic like `2 + 3`. Now give that calculator the ability to understand **locations** instead of just numbers—like "use whatever value is in slot A1." Then teach it to understand **relationships between locations**—"sum up everything from slot A1 through A10." Finally, give it the ability to **recalculate automatically** when any of those referenced locations change. That's exactly what our formula parser and evaluator does: it's a specialized calculator that speaks the language of spreadsheet addresses.

Think of the formula string as a **recipe** that specifies both the ingredients (cell references, numbers) and the cooking instructions (operators, functions). The parser's job is to read this recipe, understand the precise steps, and then execute them whenever the ingredients change. This mental model helps understand why parsing happens in two distinct phases: first **understanding** the recipe (parsing), then **executing** it (evaluation).

Parser/Evaluator Interface

The parser/evaluator component exposes a clean interface that other components (especially the dependency graph and virtual grid) can use. This separation allows the grid to display formulas, the parser to evaluate them, and the graph to track dependencies—all without tight coupling.

Table: Formula Parser Interface Methods

Method Name	Parameters	Returns	Description
`parseFormula`	`formulaString: string`, `contextCellAddress: string`	`ParsedFormula`	Main entry point: parses formula string into structured representation with resolved dependencies
`tokenize`	`input: string`	`Array<Token>`	Breaks formula into lexical tokens (numbers, operators, references, functions)
`buildAST`	`tokens: Array<Token>`, `contextCellAddress: string`	`ASTNode`	Constructs abstract syntax tree from tokens using recursive descent
`evaluateAST`	`node: ASTNode`, `valueResolver: (address: string) => any`	`any`	Recursively evaluates AST using provided resolver function to fetch cell values
`extractDependencies`	`node: ASTNode`	`Set<string>`	Traverses AST to collect all cell references (for dependency graph)
`resolveRange`	`rangeExpression: string`, `contextCellAddress: string`	`Array<string>`	Expands A1:A10 into individual cell addresses for functions like SUM

Table: ParsedFormula Data Structure

Field Name	Type	Description
`ast`	`ASTNode`	Root node of the abstract syntax tree
`dependencies`	`Set<string>`	All cell addresses referenced in the formula (for dependency tracking)
`originalString`	`string`	The original formula string (for display and editing)
`hasErrors`	`boolean`	Whether the formula contains parse or semantic errors
`errorMessage`	`string | null`	Error description if parsing failed

Table: ASTNode Types (Union Type)

Node Type	Fields	Description
`NumberNode`	`value: number`	Literal numeric value (e.g., `42`, `3.14`)
`CellReferenceNode`	`address: string`, `parsedRef: ParsedCellReference`	Reference to a single cell (e.g., `A1`, `$B$3`)
`BinaryOpNode`	`operator: '+' | '-' | '*' | '/'`, `left: ASTNode`, `right: ASTNode`	Binary operation with left and right operands
`UnaryOpNode`	`operator: '-'`, `operand: ASTNode`	Unary negation (e.g., `-A1`)
`FunctionCallNode`	`functionName: string`, `args: Array<ASTNode>`	Function call with arguments (e.g., `SUM(A1, A2)`)
`RangeNode`	`start: CellReferenceNode`, `end: CellReferenceNode`	Cell range (e.g., `A1:A10` for use in functions)
`ErrorNode`	`message: string`	Placeholder for parse errors

Parsing and Evaluation Steps

The journey from a raw formula string like `=SUM(A1:A10) * 2` to a computed numeric value involves a carefully orchestrated pipeline. Each step transforms the representation while preserving the semantic meaning.

Step 1: Formula Detection and Normalization

1. The grid component detects a formula when cell content starts with `=` (or the user explicitly marks it as a formula)
2. The leading `=` is stripped, leaving the pure expression (`SUM(A1:A10) * 2`)
3. Whitespace is normalized (optional but improves error messages and consistency)

Step 2: Lexical Analysis (Tokenization)

1. The tokenizer scans the string character by character, grouping characters into meaningful tokens
2. Different token types require different scanning rules:
   - **Numbers**: Consume digits and optional decimal point
   - **Cell references**: Start with letter(s), then digits, optionally with `$` for absolute references
   - **Operators**: Single characters like `+`, `-`, `*`, `/`
   - **Parentheses**: `(` and `)` for grouping and function calls
   - **Function names**: Letters followed by `(` (e.g., `SUM(`)
   - **Commas**: Argument separators in functions
   - **Colons**: Range operators (`A1:A10`)
3. The tokenizer must handle edge cases like negative numbers (`-5` vs subtraction `A1-5`) by examining context

Step 3: Syntax Analysis (AST Construction)

1. The parser uses a **recursive descent** approach, starting with the lowest-precedence operators (addition/subtraction)
2. The algorithm follows the classic expression grammar:

```
expression → term (('+' | '-') term)*
term → factor (('*' | '/') factor)*
factor → number | cell_reference | '(' expression ')' | function_call | '-' factor
function_call → identifier '(' arguments? ')'
arguments → expression (',' expression)*
range → cell_reference ':' cell_reference
```

3. As the parser progresses, it builds a tree where:
    - Leaf nodes are numbers or cell references
    - Internal nodes are operators or function calls
    - Parentheses create subtree boundaries
4. Operator precedence is naturally enforced by the grammar structure (multiplication before addition)

Step 4: Dependency Extraction

1. A post-order traversal of the AST collects all `CellReferenceNode` instances
2. Each reference is normalized (converted to absolute address based on context cell for relative references)
3. The set of unique dependencies is returned for the dependency graph to track

Step 5: Evaluation (When Values Are Needed)

1. The evaluator traverses the AST in post-order (children before parent)
2. For leaf nodes:
    - `NumberNode`: Return the literal value
    - `CellReferenceNode`: Call the provided `valueResolver` function to fetch current cell value
3. For internal nodes:
    - `BinaryOpNode`: Evaluate left and right children, apply operator with proper type coercion
    - `UnaryOpNode`: Evaluate operand, apply negation
    - `FunctionCallNode`: Evaluate all arguments, pass to function implementation
4. Evaluation handles errors gracefully:
    - Invalid references return `#REF!`
    - Division by zero returns `#DIV/0!`
    - Type mismatches return `#VALUE!`

Step 6: Result Formatting

1. The raw computed value may be formatted based on cell formatting rules
2. Numeric results are preserved as numbers for further calculations
3. Error results propagate upward through dependent formulas

> **Key Insight:** The separation between parsing (which happens once when formula is entered) and evaluation (which happens many times during recalculation) is crucial for performance. We pay the parsing cost upfront, then reuse the AST for fast repeated evaluations.

Architecture Decision Records for Parser

Decision: Recursive Descent Parser over Shunting Yard Algorithm

- **Context**: We need to parse arithmetic expressions with cell references, functions, and proper operator precedence. The parser must be maintainable, debuggable, and extensible for future functions.

- **Options Considered**:
  1. **Recursive descent parser**: Manually written parser following the expression grammar
  2. **Shunting yard algorithm**: Operator precedence parsing using stacks
  3. **Parser generator (PEG.js, Nearley)**: Generate parser from formal grammar
- **Decision**: Recursive descent parser
- **Rationale**:
  - Recursive descent provides excellent error messages (we can pinpoint where parsing failed)
  - It's straightforward to extend for new syntax (like ranges or new functions)
  - The code mirrors the grammar directly, making it readable and maintainable
  - No external dependencies needed (pure JavaScript)
  - While shunting yard is efficient, recursive descent handles our needs with similar performance for spreadsheet-scale formulas
- **Consequences**:
  - We must manually handle operator precedence through grammar design
  - The parser code is longer but more transparent
  - Adding new binary operators requires grammar modification

Table: Parser Approach Comparison

Option	Pros	Cons	Why Not Chosen
Recursive Descent	Excellent error messages, readable, extensible	Manual implementation, longer code	**Chosen** for maintainability and clarity
Shunting Yard	Efficient, handles arbitrary precedence	Harder to debug, less intuitive for functions	Good for pure arithmetic but less flexible for spreadsheet syntax
Parser Generator	Formal grammar specification, generated code	External dependency, learning curve, build step	Overkill for our relatively simple grammar

Decision: AST-Based Evaluation over Direct Interpretation

- **Context**: After parsing a formula, we need to evaluate it repeatedly (when dependencies change). We must choose between interpreting the tokens directly or building an intermediate tree representation.
- **Options Considered**:
  1. **AST evaluation**: Build complete abstract syntax tree, then evaluate it
  2. **Direct interpretation**: Evaluate during parsing using value stack
  3. **Bytecode compilation**: Compile to simple bytecode for faster evaluation
- **Decision**: AST-based evaluation
- **Rationale**:
  - AST separates parsing concerns from evaluation concerns cleanly
  - The same AST can be used for multiple purposes: evaluation, dependency extraction, formula display
  - ASTs are easier to debug and visualize (we can "see" the formula structure)
  - Performance difference is negligible for typical spreadsheet formulas (rarely more than 100 nodes)
  - Bytecode compilation adds significant complexity for minimal gain in our context
- **Consequences**:
  - We maintain two data structures: tokens and AST
  - Memory overhead for storing ASTs in many cells

- Evaluation requires tree traversal (slightly slower than stack machine)

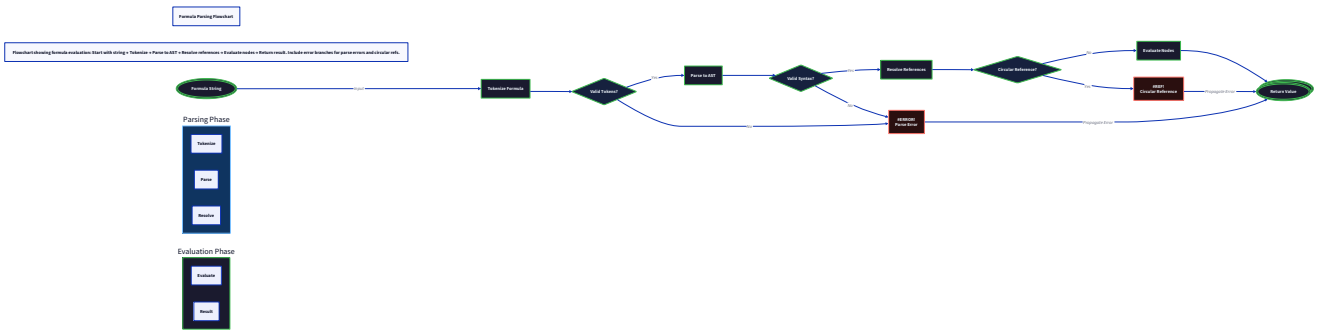
Decision: Lazy Dependency Extraction

- **Context**: When a formula is parsed, we need to know which cells it references to build the dependency graph. We must decide when to extract these dependencies.
- **Options Considered**:
  1. **Eager extraction**: Extract dependencies during parsing
  2. **Lazy extraction**: Extract dependencies via separate AST traversal after parsing
  3. **Hybrid**: Track dependencies during parsing but verify later
- **Decision**: Lazy extraction via separate traversal
- **Rationale**:
  - Separation of concerns: parsing creates the AST, dependency extraction analyzes it
  - We might need dependencies at different times (immediately for graph, later for UI highlighting)
  - The extraction algorithm is simple (tree traversal) and can be tested independently
  - If parsing fails, we skip extraction entirely (no partial dependency tracking)
- **Consequences**:
  - Small performance penalty (traversing tree twice: once for parsing, once for extraction)
  - Cleaner code organization with single-responsibility functions

Common Pitfalls: Formula Parsing

⚠️ Pitfall 1: Ignoring Operator Precedence

- **Description**: Implementing evaluation as simple left-to-right without respecting `*` and `/` before `+` and `-`
- **Why it's wrong**: `1 + 2 * 3` would evaluate to `9` instead of `7`, breaking standard mathematical expectations
- **How to fix**: Use proper expression grammar with precedence levels or shunting yard algorithm



⚠️ Pitfall 2: Mishandling Negative Numbers

- **Description**: Treating the `-` in `-5` as a binary operator with missing left operand instead of a unary operator
- **Why it's wrong**: Parser gets confused about whether `-` is subtraction or negation, leading to parse errors
- **How to fix**: In the grammar, define unary negation at the `factor` level: `factor → ('+' | '-')? factor | primary`

⚠️ Pitfall 3: Not Validating Cell References

- **Description**: Accepting invalid cell addresses like `A0`, `AA`, or `A1001` (beyond MAX_ROWS)
- **Why it's wrong**: References to non-existent cells cause evaluation errors and break the dependency graph
- **How to fix**: Use `isValidCellAddress()` helper when parsing references, mark formula with `#REF!` error

⚠️ Pitfall 4: Forgetting About Absolute References

- **Description**: Treating `$A$1` the same as `A1`, losing the absolute reference semantics
- **Why it's wrong**: Copy-paste with reference adjustment will break, as absolute references shouldn't change
- **How to fix**: Store `rowAbsolute` and `columnAbsolute` flags in `ParsedCellReference`, preserve them in AST

⚠️ Pitfall 5: Infinite Recursion with Circular References

- **Description**: Parser handles `A1` referencing `A1` directly or through chain `A1→B1→A1`
- **Why it's wrong**: Evaluation enters infinite loop, browser hangs
- **How to fix**: Dependency graph detects cycles before evaluation; evaluator should have recursion depth limit

⚠️ Pitfall 6: Case Sensitivity in Function Names

- **Description**: Treating `SUM`, `sum`, and `Sum` as different functions
- **Why it's wrong**: Spreadsheet users expect case-insensitive function names (Excel convention)
- **How to fix**: Normalize function names to uppercase during tokenization

⚠️ Pitfall 7: Not Handling Empty Cells

- **Description`: Treating references to empty cells as errors or 0` inconsistently
- **Why it's wrong**: `SUM(A1, A2)` where A1 is empty should treat A1 as `0`, not error
- **How to fix`: Value resolver should return 0 or empty string based on context; functions should handle` undefined`

Implementation Guidance: Parser Component

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
Tokenization	Manual character-by-character scanner	Regular expressions with capture groups
AST Representation	Plain JavaScript objects with `type` field	Class hierarchy with visitor pattern
Error Handling	Try-catch with custom error types	Parser combinator library (like `parsimmon`)
Function Library	Hardcoded JavaScript functions	Pluggable function registry with validation

B. Recommended File/Module Structure

```
src/
├── formula/
│   ├── parser/
│   │   ├── index.js           # Main exports: parseFormula, evaluateFormula
│   │   ├── tokenizer.js       # Token types, tokenize() function
│   │   ├── ast-builder.js     # buildAST(), AST node types
│   │   ├── evaluator.js       # evaluateAST(), function implementations
│   │   ├── dependency-extractor.js  # extractDependencies()
│   │   └── utils.js          # isFormulaString(), normalizeFormula()
│   ├── functions/
│   │   ├── index.js           # Function registry: SUM, AVG, MIN, MAX, COUNT
│   │   └── math.js           # Mathematical function implementations
│   └── errors.js              # Formula error constants and helpers
```

C. Infrastructure Starter Code (Complete)

JAVASCRIPT

```javascript
// src/formula/errors.js

export const ERRORS = {

  CIRCULAR_REF: '#CIRCULAR_REF!',

  REF_ERROR: '#REF!',

  VALUE_ERROR: '#VALUE!',

  DIV_ZERO: '#DIV/0!',

  PARSE_ERROR: '#ERROR!',

  NAME_ERROR: '#NAME?',

  NUM_ERROR: '#NUM!',

  NA_ERROR: '#N/A'

};


// Helper to check if a value is an error

export function isError(value) {

  return typeof value === 'string' && value.startsWith('#') && value.endsWith('!');

}


// src/formula/utils.js

export function isFormulaString(str) {

  return typeof str === 'string' && str.trim().startsWith('=');

}


export function normalizeFormula(formula) {

  // Remove leading = and trim whitespace

  const trimmed = formula.trim();

  return trimmed.startsWith('=') ? trimmed.substring(1).trim() : trimmed;

}


// src/formula/parser/tokenizer.js

export const TokenType = {

  NUMBER: 'NUMBER',

  CELL_REF: 'CELL_REF',

  OPERATOR: 'OPERATOR',

  FUNCTION: 'FUNCTION',
```

```
  LEFT_PAREN: 'LEFT_PAREN',
  RIGHT_PAREN: 'RIGHT_PAREN',
  COMMA: 'COMMA',
  COLON: 'COLON',
  EOF: 'EOF'
};

export class Token {
  constructor(type, value, position) {
    this.type = type;
    this.value = value;
    this.position = position; // For error reporting
  }
}

// Simple scanner implementation
export function scanFormula(input) {
  const tokens = [];
  let pos = 0;
  const inputLength = input.length;

  while (pos < inputLength) {
    const char = input[pos];
    const startPos = pos;

    // Skip whitespace
    if (/\s/.test(char)) {
      pos++;
      continue;
    }

    // Numbers
    if (/[0-9]/.test(char) || (char === '.' && /[0-9]/.test(input[pos + 1]))) {
```

```
    let numStr = '';
    while (pos < inputLength && (/[0-9]/.test(input[pos]) || input[pos] === '.')) {
      numStr += input[pos];
      pos++;
    }
    tokens.push(new Token(TokenType.NUMBER, parseFloat(numStr), startPos));
    continue;
  }

  // Cell references: letters then digits, optionally with $ signs
  if (/[A-Za-z$]/.test(char)) {
    let refStr = '';
    // Parse column part (letters, optionally with $)
    while (pos < inputLength && /[A-Za-z$]/.test(input[pos])) {
      refStr += input[pos];
      pos++;
    }
    // Parse row part (digits, optionally with $)
    while (pos < inputLength && /[0-9$]/.test(input[pos])) {
      refStr += input[pos];
      pos++;
    }

    // Check if it's a function (letters followed by '(')
    const nextChar = pos < inputLength ? input[pos] : '';
    if (/[A-Za-z]/.test(char) && nextChar === '(') {
      tokens.push(new Token(TokenType.FUNCTION, refStr.toUpperCase(), startPos));
    } else {
      tokens.push(new Token(TokenType.CELL_REF, refStr, startPos));
    }
    continue;
  }
```

```
    // Operators
    if ('+-*/'.includes(char)) {
      tokens.push(new Token(TokenType.OPERATOR, char, startPos));
      pos++;
      continue;
    }

    // Parentheses and separators
    if (char === '(') {
      tokens.push(new Token(TokenType.LEFT_PAREN, char, startPos));
      pos++;
      continue;
    }

    if (char === ')') {
      tokens.push(new Token(TokenType.RIGHT_PAREN, char, startPos));
      pos++;
      continue;
    }

    if (char === ',') {
      tokens.push(new Token(TokenType.COMMA, char, startPos));
      pos++;
      continue;
    }

    if (char === ':') {
      tokens.push(new Token(TokenType.COLON, char, startPos));
      pos++;
      continue;
    }
```

```
    // Unknown character
    throw new Error(`Unexpected character '${char}' at position ${pos}`);
  }

  tokens.push(new Token(TokenType.EOF, '', pos));
  return tokens;
}
```

D. Core Logic Skeleton Code

JAVASCRIPT

```javascript
// src/formula/parser/ast-builder.js

import { TokenType } from './tokenizer.js';

import { parseCellAddress, isValidCellAddress } from '../../utils/cell-utils.js';


// AST node type definitions (as plain objects)

export function createNumberNode(value) {

  return { type: 'NumberNode', value };

}


export function createCellReferenceNode(address, parsedRef) {

  return { type: 'CellReferenceNode', address, parsedRef };

}


export function createBinaryOpNode(operator, left, right) {

  return { type: 'BinaryOpNode', operator, left, right };

}


export function createFunctionCallNode(functionName, args) {

  return { type: 'FunctionCallNode', functionName, args };

}


// Recursive descent parser

export class FormulaParser {

  constructor(tokens) {

    this.tokens = tokens;

    this.position = 0;

    this.currentToken = this.tokens[0];

  }



  // Main entry point: parse expression

  parseExpression() {

    // TODO 1: Start with addition/subtraction level (lowest precedence)

    // TODO 2: Call parseTerm() to get left operand

    // TODO 3: While current token is '+' or '-', consume token and parse right term

    // TODO 4: Build binary operation node combining left and right
```

```
  // TODO 5: Return the resulting AST node
}

parseTerm() {
  // TODO 1: Parse factor as left operand
  // TODO 2: While current token is '*' or '/', consume token and parse right factor
  // TODO 3: Build binary operation node for multiplication/division
  // TODO 4: Return the resulting node
}

parseFactor() {
  // TODO 1: Handle unary plus/minus: if token is '+' or '-', consume and parse factor
  // TODO 2: Check token type:
  //   - NUMBER: consume and return NumberNode
  //   - CELL_REF: parse address, validate, return CellReferenceNode
  //   - LEFT_PAREN: consume '(', parse expression, expect ')'
  //   - FUNCTION: parse function call
  // TODO 3: Throw error for unexpected token
}

parseFunctionCall() {
  // TODO 1: Current token should be FUNCTION type, get function name
  // TODO 2: Consume '(' token
  // TODO 3: Parse arguments: while not RIGHT_PAREN or EOF
  //   - Parse expression as argument
  //   - If next token is COMMA, consume and continue
  // TODO 4: Expect ')' token
  // TODO 5: Return FunctionCallNode with function name and arguments array
}

// Helper methods
advance() {
```

```
    this.position++;
    this.currentToken = this.tokens[this.position] || { type: TokenType.EOF, value: '' };
  }

  expect(tokenType, errorMessage) {
    if (this.currentToken.type !== tokenType) {
      throw new Error(errorMessage);
    }
    const token = this.currentToken;
    this.advance();
    return token;
  }
}

// src/formula/parser/evaluator.js
export function evaluateNode(node, valueResolver) {
  // TODO 1: Handle each node type:
  //   - NumberNode: return node.value
  //   - CellReferenceNode: call valueResolver(node.address), handle errors
  //   - BinaryOpNode: evaluate left and right, apply operator with type checking
  //   - FunctionCallNode: evaluate all arguments, call function implementation
  // TODO 2: For binary operators, handle numeric coercion (convert strings to numbers)
  // TODO 3: For division, check for division by zero and return DIV_ZERO error
  // TODO 4: Propagate errors: if any operand is error, return that error
}

// src/formula/parser/dependency-extractor.js
export function extractDependencies(node) {
  const dependencies = new Set();

  function traverse(currentNode) {
    // TODO 1: For CellReferenceNode: add node.address to dependencies set
    // TODO 2: For BinaryOpNode: traverse left and right children
```

```
    // TODO 3: For FunctionCallNode: traverse all arguments
    // TODO 4: For RangeNode: expand range to individual cells, add all to dependencies
    // TODO 5: Ignore NumberNode and ErrorNode
  }

  traverse(node);
  return dependencies;
}

// src/formula/parser/index.js
export function parseFormula(formulaString, contextCellAddress) {
  try {
    // TODO 1: Normalize formula string (remove '=', trim)
    // TODO 2: Tokenize using scanFormula()
    // TODO 3: Parse tokens into AST using FormulaParser
    // TODO 4: Extract dependencies from AST
    // TODO 5: Return ParsedFormula object with AST, dependencies, original string
  } catch (error) {
    // TODO 6: On parse error, return ParsedFormula with hasErrors: true and errorMessage
  }
}
```

E. Language-Specific Hints

1. **Use Iterators for Token Stream**: Instead of array indexing, implement a token iterator with `peek()` and `consume()` methods for cleaner parser code.
2. **Immutable AST Nodes**: Create AST nodes as frozen objects or using `Object.freeze()` to prevent accidental mutation during evaluation.
3. **Memoization for Performance**: Cache parsed formulas in a weak map keyed by formula string and context cell to avoid re-parsing identical formulas.
4. **Error Recovery in Parser**: Implement "panic mode" error recovery by skipping tokens until a known synchronizing point (like `,` or `)`).
5. **Use Generators for Range Expansion**: When expanding `A1:A10`, use a generator function to yield individual addresses without creating large intermediate arrays.

F. Milestone Checkpoint

After implementing the formula parser component, verify it works correctly:

1. **Unit Tests**: Run parser tests with `npm test` or `node test-parser.js`

```
Expected output: All tests pass with green checkmarks
```

2. **Manual Verification**:
   - Enter `=2+3*4` in a cell - should display `14` (not `20`)
   - Enter `=SUM(A1:A3)` where A1=1, A2=2, A3=3 - should display `6`
   - Enter `=A1+B1` where A1=5, B1=10 - should display `15`
   - Enter invalid formula `=2+*3` - should show `#ERROR!`

3. **Debug Visualization**:

```javascript
// Add this temporary debug function
function debugParse(formula) {
  const result = parseFormula(formula, 'A1');
  console.log('AST:', JSON.stringify(result.ast, null, 2));
  console.log('Dependencies:', Array.from(result.dependencies));
}
debugParse('=SUM(A1:B2)*2');
```

4. **Signs Something Is Wrong**:
   - Formulas with parentheses don't evaluate correctly
   - Operator precedence is wrong (`1+2*3` gives `9`)
   - Cell references to empty cells cause errors instead of treating as `0`
   - Functions are case-sensitive

Component Design: Dependency Graph and Recalculation Engine

> **Milestone(s):** Milestone 3 (Dependency Graph & Recalculation)

The dependency graph and recalculation engine form the computational heart of the spreadsheet application. While the virtualized grid handles presentation and the formula parser converts strings to executable expressions, this component manages the reactive dataflow that makes spreadsheets feel "alive"—changing one cell automatically updates all cells that depend on it. This section details the directed graph data structure that tracks cell relationships, the topological sort algorithm that ensures correct evaluation order, and the mechanisms that prevent infinite loops from circular references.

Mental Model: A Network of Data Dependencies

Imagine a spreadsheet as a network of interconnected calculators, each with its own display. When you change the input of one calculator, the change propagates through the network like ripples in water—each calculator that depends on the changed value automatically recalculates, which triggers its own dependents, and so on. The critical insight is that this propagation must follow the **direction of data flow**: a cell should only recalculate after all the cells it references have been updated.

Think of each cell as a **node** in a directed graph, with arrows pointing **from a referenced cell to the cell that references it**. If cell B2 contains the formula `=A1+5`, then an arrow points from A1 to B2 (B2 depends on A1). This direction is crucial: when A1

changes, we follow the arrows forward to find what needs updating. The graph must remain **acyclic** (no circular paths) to avoid infinite recalculation loops. When you type `=B2` into A1, you're creating a cycle (A1→B2→A1), which the system must detect and prevent.

Another useful analogy is a **recipe dependency graph** in cooking. To bake a cake (cell C), you need eggs (cell A) and flour (cell B). You must obtain eggs and flour before you can mix them. If the cake recipe itself were listed as an ingredient for making eggs, you'd have a circular dependency that makes no sense. The spreadsheet's recalculation engine ensures all "ingredients" (referenced cells) are prepared before "mixing" (evaluating the formula).

Dependency Graph Interface

The `DependencyGraph` is the central data structure that tracks relationships between cells. It maintains two core mappings: from cell addresses to their data (`nodes`), and from cell addresses to their dependents (`edges`). This **adjacency list** representation efficiently supports the operations needed for recalculation: adding/removing dependencies, finding all cells that depend on a given cell, and detecting cycles.

The following table details the complete interface for the `DependencyGraph` :

Method Signature	Parameters	Returns	Description
`addDependency(dependentAddress, dependencyAddress)`	`dependentAddress: string`, `dependencyAddress: string`	`void`	Establishes that `dependentAddress` cell's value depends on `dependencyAddress`. Updates the adjacency list and validates that this doesn't create a circular reference.
`removeDependency(dependentAddress, dependencyAddress)`	`dependentAddress: string`, `dependencyAddress: string`	`void`	Removes the dependency relationship between the two cells. Called when a cell's formula changes or is cleared.
`getDependents(cellAddress)`	`cellAddress: string`	`Set<string>`	Returns all cells that directly depend on the given cell (its immediate "children" in the dependency graph). Essential for propagating changes.
`getTopologicalOrder(changedCell)`	`changedCell: string`	`string[]`	Given a changed cell address, returns an array of cell addresses in topological order (dependencies before dependents) for all cells that need recalculation.
`detectCircularReference(startAddress, targetAddress)`	`startAddress: string`, `targetAddress: string`	`boolean`	Performs a depth-first search to determine if adding an edge from `startAddress` to `targetAddress` would create a cycle. Returns `true` if a cycle would be created.
`clearDependenciesForCell(cellAddress)`	`cellAddress: string`	`void`	Removes all incoming and outgoing dependencies for a cell. Used when a cell's content is cleared or replaced with a non-formula value.
`hasDependencies(cellAddress)`	`cellAddress: string`	`boolean`	Checks whether the given cell has any outgoing dependencies (references other cells). Useful for optimization.

The `DependencyGraph` works in tandem with the `Spreadsheet` data model, which stores the actual cell values and formulas. When a cell's formula is parsed by the `FormulaParser`, the resulting `ParsedFormula` includes a `dependencies` field—a `Set<string>` of cell addresses referenced in the formula. The graph component compares this new dependency set with the cell's previous dependencies, adding and removing edges as needed.

Recalculation Algorithm with Topological Sort

When a cell's value changes (either directly or through formula recalculation), the spreadsheet must efficiently update all cells that depend on it, directly or indirectly. The recalculation algorithm follows a systematic five-step process:

1. **Dirty Marking**: When cell `C` changes, mark `C` and all cells reachable from `C` via dependency edges as "dirty" (needing recalculation). This forms the **affected set**.
2. **Topological Sorting**: Perform a topological sort on the subgraph consisting of dirty cells and their dependencies. This ensures cells are evaluated in an order where no cell is calculated before its dependencies.
3. **Cycle Safety Check**: Before attempting topological sort, verify the entire graph remains acyclic. If a circular reference is detected (should have been caught earlier, but defensive check), abort and show error.
4. **Sequential Evaluation**: Evaluate each cell in the topological order. For formula cells, retrieve current values of referenced cells (which are guaranteed to be up-to-date due to the sort order) and compute new value.
5. **Update Propagation**: After evaluation, if a cell's value actually changed, propagate this change by marking its dependents as dirty and repeating the process (though typically handled in one pass with proper topological ordering).

The topological sort algorithm uses **Kahn's Algorithm**, which is particularly suitable because it naturally detects cycles and works efficiently with adjacency lists:

Algorithm Steps:

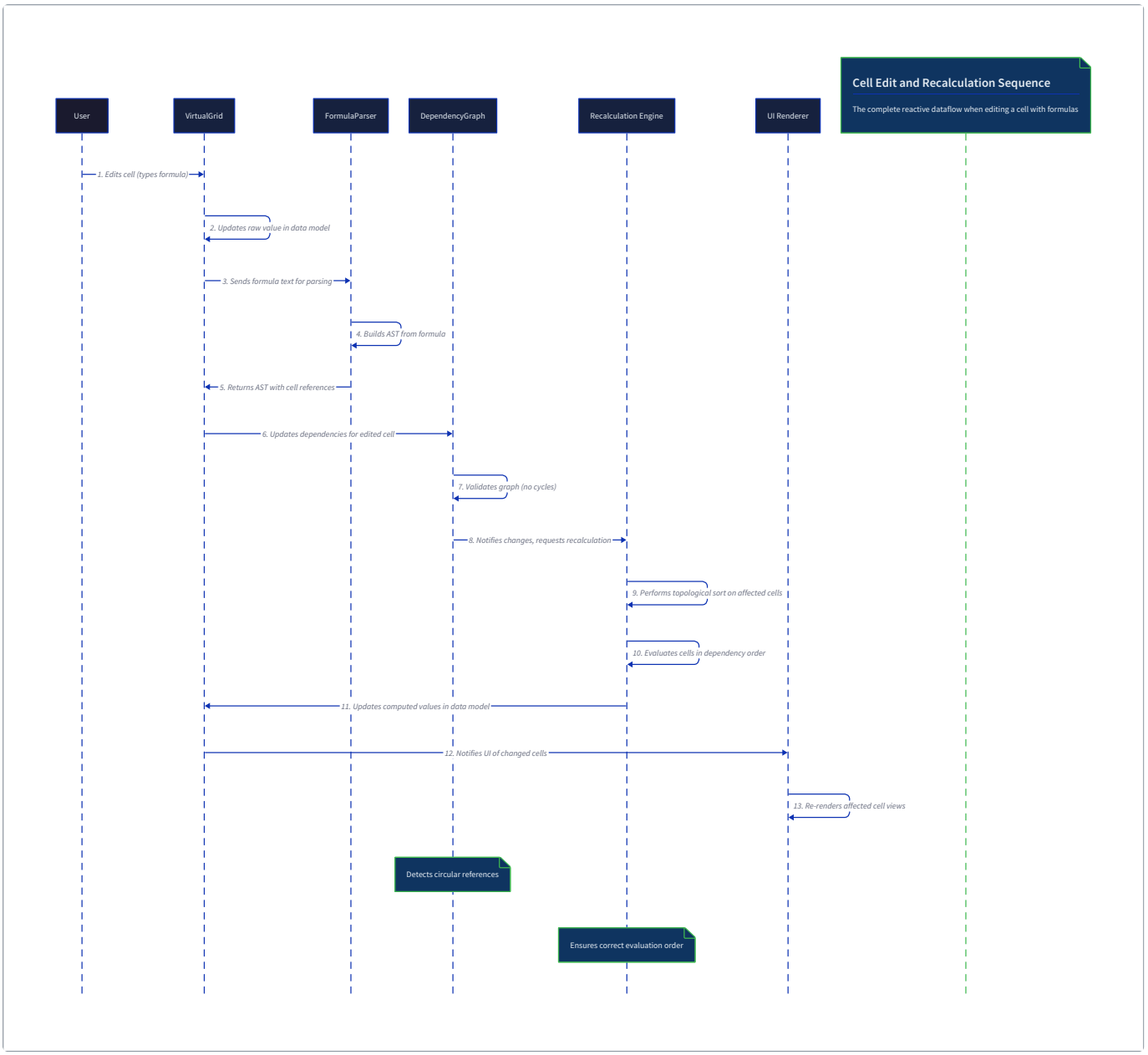
1. Compute **in-degree** for each cell in the affected set (number of incoming edges from other cells in the affected set).
2. Initialize a queue with all cells having in-degree 0 (cells with no dependencies within the affected set, or whose dependencies are already up-to-date).
3. While the queue is not empty:
   - Remove a cell from the queue and add it to the topological order.
   - For each dependent of this cell (from `getDependents`):
     - Decrement the dependent's in-degree.
     - If the dependent's in-degree becomes 0, add it to the queue.
4. If the topological order contains fewer cells than the affected set, a cycle exists within the affected subgraph.

> **Key Insight:** The recalculation engine is **reactive but lazy**—it doesn't recalculate the entire spreadsheet on every change, only the transitive closure of cells affected by the change. This incremental approach is essential for performance with large spreadsheets.

The following state machine describes the dependency graph's behavior during the recalculation process:

Current State	Event	Next State	Actions Taken
Normal	Cell value changes	Adding Edge	Parse formula to extract dependencies, call `addDependency` for each
Adding Edge	No circular reference detected	Normal	Edge added to adjacency list, cell marked for recalculation
Adding Edge	Circular reference detected	Error State	Edge rejected, error displayed in cell, graph unchanged
Normal	Recalculation triggered	Recalculating	Compute affected set, perform topological sort
Recalculating	Topological sort succeeds	Updating	Evaluate cells in sorted order, update values
Updating	All cells evaluated	Normal	Clear dirty flags, trigger UI updates
Recalculating	Cycle detected in affected set	Error State	Show circular reference error, abort recalculation
Error State	User fixes formula	Normal	Clear error, restart dependency analysis

The sequence diagram below illustrates the complete flow when a user edits a cell containing a formula:


Cell Edit and Recalculation Sequence
The complete reactive dataflow when editing a cell with formulas
User
VirtualGrid
FormulaParser
DependencyGraph
Recalculation Engine
UI Renderer
1. Edits cell (types formula)
2. Updates raw value in data model
3. Sends formula text for parsing
4. Builds AST from formula
5. Returns AST with cell references
6. Updates dependencies for edited cell
7. Validates graph (no cycles)
8. Notifies changes, requests recalculation
9. Performs topological sort on affected cells
10. Evaluates cells in dependency order
11. Updates computed values in data model
12. Notifies UI of changed cells
13. Re-renders affected cell views
Detects circular references
Ensures correct evaluation order

Architecture Decision Records for Dependency Graph

Decision: Adjacency List Representation for Dependency Graph

- **Context**: We need to track which cells depend on which other cells to enable efficient recalculation when source cells change. The graph must support adding/removing edges, finding all dependents of a cell, and cycle detection.
- **Options Considered**:
  1. **Adjacency Matrix**: 2D boolean array where `matrix[i][j]` indicates cell `i` depends on cell `j`.
  2. **Adjacency List**: Map from cell addresses to sets of dependent addresses (forward edges) or referenced addresses (backward edges).
  3. **Implicit Graph via Cell Objects**: Each `Cell` object stores its dependencies and dependents as properties.
- **Decision**: Adjacency list storing forward edges (dependents).
- **Rationale**: The adjacency list provides optimal time complexity for our primary operations: finding all dependents of a cell is O(1) with a set lookup; adding/removing edges is O(1). It uses memory proportional to the number of edges rather than $O(n^2)$ like a matrix. Storing forward edges (dependents) rather than backward edges (dependencies) simplifies the propagation of changes from a modified cell. The graph is maintained separately from cell objects to maintain separation of concerns—cells don't need to know about their dependents.
- **Consequences**: We need to update the graph whenever formulas change. Cycle detection requires traversal algorithms. The graph must be kept in sync with cell formulas.

Option	Pros	Cons	Chosen?
Adjacency Matrix	Simple implementation, cycle detection via matrix exponentiation	Memory $O(n^2)$ for n cells, sparse graphs waste space	No
Adjacency List	**Memory proportional to edges, fast dependent lookup, standard graph algorithms**	**Requires explicit graph maintenance**	**Yes**
Implicit Graph in Cells	No separate data structure, dependencies stored with cell	Hard to traverse in reverse (find dependents), cells coupled to graph logic	No

Decision: Kahn's Algorithm for Topological Sort

- **Context**: After a cell change, we need to recalculate dependent cells in correct order (dependencies before dependents). We must handle potentially large affected subgraphs efficiently and detect cycles.
- **Options Considered**:
  1. **Depth-First Search (DFS) with timestamps**: Recursive algorithm that produces reverse postorder.
  2. **Kahn's Algorithm (BFS-based)**: Iteratively removes nodes with zero in-degree.
  3. **Manual dependency levels**: Assign each cell a "depth" level and sort by level.
- **Decision**: Kahn's Algorithm.
- **Rationale**: Kahn's algorithm naturally detects cycles (if queue empties before all nodes are processed, a cycle exists). It's intuitive to implement and debug—you can visualize "peeling away" layers of dependencies. The BFS nature tends to recalculate "closer" dependents first, which can provide better visual feedback. Performance is O(V+E), same as DFS, but without recursion depth limits.
- **Consequences**: Requires maintaining in-degree counts for nodes, which we can compute on-the-fly for the affected subgraph. We need to rebuild the in-degree map for each recalculation (acceptable since affected subgraph is typically small).

Option	Pros	Cons	Chosen?
DFS with timestamps	Classical algorithm, can detect cycles with three-color method	Recursion depth limits, less intuitive ordering	No
Kahn's Algorithm	**Cycle detection built-in, intuitive "layer-by-layer" processing, no recursion**	**Requires in-degree computation**	**Yes**
Manual dependency levels	Simple to implement, constant-time level lookup	Hard to maintain when formulas change, doesn't handle all DAG structures	No

Decision: Immediate Cycle Detection on Edge Addition

- **Context**: When a user enters a formula that creates a circular reference (A1 references B1, B1 references A1), we must prevent the invalid state from entering the graph.
- **Options Considered**:
  1. **Preventive detection**: Check for cycles when adding each dependency edge, reject if cycle would form.
  2. **Lazy detection**: Allow the edge, detect cycles during topological sort, then roll back.
  3. **Timeout-based detection**: Allow the edge, attempt recalculation, timeout if takes too long (suggesting cycle).
- **Decision**: Preventive detection on edge addition.
- **Rationale**: It's better to fail fast and give immediate feedback than to allow an invalid state that might cause confusing behavior later. Preventive detection localizes the error to the specific formula being edited. The computational cost is acceptable because dependency chains in spreadsheets are typically shallow, and we only check the subgraph reachable from the new dependency.
- **Consequences**: Need to implement `detectCircularReference` that performs DFS from the prospective dependency to see if it reaches the dependent cell. Users get immediate error feedback when creating circular references.

Common Pitfalls: Dependency Management

⚠️ Pitfall: Forgetting to Clear Old Dependencies

- **Description**: When a cell's formula changes from `=A1+B1` to `=C1+D1`, developers often add the new dependencies (C1, D1) but forget to remove the old ones (A1, B1). The graph retains stale edges.
- **Why it's wrong**: The cell will incorrectly recalculate when A1 or B1 changes, even though it no longer references them. This wastes computation and can cause incorrect values if A1/B1 are also referenced elsewhere in surprising ways.
- **How to fix**: Always compute the **delta** between old and new dependency sets. Use `removeDependency` for dependencies in old set but not new set; use `addDependency` for dependencies in new set but not old set.

⚠️ Pitfall: Incorrect Edge Direction

- **Description**: Creating edges from dependency to dependent (`A1 → B1` when B1 depends on A1) instead of from dependent to dependency (`B1 → A1`).
- **Why it's wrong**: Topological sort and dependency propagation algorithms assume edges point in the direction of data flow (from prerequisite to dependent). Reversed edges make it hard to find all cells affected by a change.
- **How to fix**: Maintain consistent mental model: "An edge from X to Y means Y depends on X" or "Y references X". Document this convention and write tests that verify `getDependents(A1)` returns `B1` when B1 has formula `=A1`.

⚠️ Pitfall: Not Handling Empty/Missing Cells in Dependency Resolution

- **Description**: When evaluating `=A1+B1`, if A1 is empty (has no `Cell` object), the evaluator might throw an error or treat it as undefined rather than 0.

- **Why it's wrong**: Excel and other spreadsheets treat empty cells as 0 in arithmetic operations. Failing to handle this breaks common spreadsheet patterns.
- **How to fix**: In the value resolver function passed to `evaluateAST`, return `0` (or empty string for string context) for missing cells. Ensure the `getCell` method returns a default empty cell object for uninitialized coordinates.

⚠️ **Pitfall: Recalculating the Entire Spreadsheet**

- **Description**: On every cell change, traversing all cells with formulas and recalculating them regardless of whether they're affected by the change.
- **Why it's wrong**: Performance degrades exponentially with spreadsheet size. A 1000×1000 grid with 10% formulas would recalculate 100,000 cells for every change.
- **How to fix**: Implement true incremental recalculation using the dependency graph. Only recalculate the **transitive closure** of cells reachable from the changed cell via dependency edges. Use `getTopologicalOrder` limited to the affected subgraph.

⚠️ **Pitfall: Infinite Loop from Indirect Circular References**

- **Description**: Detecting direct circular references (A1 references B1, B1 references A1) but missing longer cycles (A1→B1→C1→A1).
- **Why it's wrong**: The spreadsheet will hang in infinite recalculation or produce stack overflow errors during evaluation.
- **How to fix**: Use proper cycle detection that traverses the entire dependency path, not just immediate dependencies. The `detectCircularReference` method should perform DFS from the target cell through all its dependents to see if it reaches the start cell.

Implementation Guidance: Graph Component

Technology Recommendations Table:

Component	Simple Option	Advanced Option
Graph Data Structure	Plain JavaScript Maps and Sets	Immutable.js for persistent data structures
Cycle Detection	Depth-First Search with recursion	Iterative DFS with explicit stack
Topological Sort	Kahn's algorithm with queue	DFS-based with postorder numbering
Performance Optimization	Basic affected subgraph computation	Memoization of evaluation results

Recommended File/Module Structure:

```
spreadsheet-engine/
├── src/
│   ├── core/
│   │   ├── dependency-graph.js      # DependencyGraph class (this component)
│   │   ├── formula-parser.js        # From previous milestone
│   │   └── spreadsheet-model.js     # Spreadsheet data model
│   ├── utils/
│   │   ├── cell-address.js          # Address parsing utilities
│   │   └── topological-sort.js      # Generic topological sort
│   └── index.js                     # Main application entry
```

Infrastructure Starter Code:

The following utility functions provide complete implementations for common operations that aren't the core learning focus but are necessary prerequisites:

JAVASCRIPT

```javascript
// src/utils/cell-address.js

/**
 * Utility functions for working with spreadsheet cell addresses.
 * Implements the exact naming conventions specified in the design document.
 */

const COLUMN_LETTERS = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ';
const MAX_ROWS = 1000;
const MAX_COLUMNS = 26;

/**
 * Parse Excel-style cell address into structured format
 * @param {string} address - Cell address like "A1", "$B$2", "C$3"
 * @returns {ParsedCellReference} Parsed reference with row/column indices and absolute flags
 */
function parseCellAddress(address) {
  if (!address || typeof address !== 'string') {
    throw new Error(`Invalid cell address: ${address}`);
  }

  // Regex matches: optional $, column letters, optional $, row number
  const match = address.match(/^(\$?)([A-Z]+)(\$?)(\d+)$/i);
  if (!match) {
    throw new Error(`Invalid cell address format: ${address}`);
  }

  const [, colAbs, colLetters, rowAbs, rowNum] = match;
  const column = letterToColumn(colLetters.toUpperCase());
  const row = parseInt(rowNum, 10) - 1; // Convert to 0-based

  if (row < 0 || row >= MAX_ROWS || column < 0 || column >= MAX_COLUMNS) {
    throw new Error(`Cell address out of bounds: ${address}`);
  }
```

```
  return {
    row,
    column,
    rowAbsolute: rowAbs === '$',
    columnAbsolute: colAbs === '$'
  };
}

/**
 * Convert row/column indices to cell address string
 * @param {number} row - 0-based row index
 * @param {number} column - 0-based column index
 * @param {boolean} rowAbsolute - Whether row is absolute reference
 * @param {boolean} columnAbsolute - Whether column is absolute reference
 * @returns {string} Excel-style cell address
 */
function toCellAddress(row, column, rowAbsolute = false, columnAbsolute = false) {
  if (row < 0 || row >= MAX_ROWS || column < 0 || column >= MAX_COLUMNS) {
    throw new Error(`Row or column out of bounds: row=${row}, col=${column}`);
  }

  const colPart = columnToLetter(column);
  const rowPart = (row + 1).toString();

  return `${columnAbsolute ? '$' : ''}${colPart}${rowAbsolute ? '$' : ''}${rowPart}`;
}

/**
 * Check if string is valid cell address
 * @param {string} address - String to validate
 * @returns {boolean} True if valid cell address
 */
```

```
function isValidCellAddress(address) {
  try {
    parseCellAddress(address);
    return true;
  } catch {
    return false;
  }
}

/**
 * Convert 0-based column index to Excel column letters
 * @param {number} column - 0-based column index (0 = A, 25 = Z)
 * @returns {string} Excel column letters
 */
function columnToLetter(column) {
  if (column < 0 || column >= MAX_COLUMNS) {
    throw new Error(`Column index out of bounds: ${column}`);
  }

  // Simple case: single letter for columns 0-25
  if (column < 26) {
    return COLUMN_LETTERS[column];
  }

  // For multi-letter columns (beyond Z), though our MAX_COLUMNS is 26
  // This handles potential future expansion
  let result = '';
  let n = column + 1; // Convert to 1-based

  while (n > 0) {
    const remainder = (n - 1) % 26;
    result = COLUMN_LETTERS[remainder] + result;
```

```
    n = Math.floor((n - 1) / 26);
  }

  return result;
}

/**
 * Convert Excel column letters to 0-based index
 * @param {string} letters - Excel column letters (e.g., "A", "AB", "ZZ")
 * @returns {number} 0-based column index
 */
function letterToColumn(letters) {
  if (!letters || !/^[A-Z]+$/i.test(letters)) {
    throw new Error(`Invalid column letters: ${letters}`);
  }

  const upperLetters = letters.toUpperCase();
  let result = 0;

  for (let i = 0; i < upperLetters.length; i++) {
    const charValue = upperLetters.charCodeAt(i) - 'A'.charCodeAt(0) + 1;
    result = result * 26 + charValue;
  }

  return result - 1; // Convert to 0-based
}

/**
 * Error constants for spreadsheet operations
 */
const ERRORS = {
  CIRCULAR_REF: '#CIRCULAR!',
  REF_ERROR: '#REF!',
```

```
  VALUE_ERROR: '#VALUE!',
  DIV_ZERO: '#DIV/0!',
  PARSE_ERROR: '#ERROR!'
};

export {
  parseCellAddress,
  toCellAddress,
  isValidCellAddress,
  columnToLetter,
  letterToColumn,
  ERRORS,
  COLUMN_LETTERS,
  MAX_ROWS,
  MAX_COLUMNS
};
```

Core Logic Skeleton Code:

The following skeleton provides the structure for the `DependencyGraph` class with detailed TODO comments mapping to the algorithm steps described earlier:

JAVASCRIPT

```javascript
// src/core/dependency-graph.js

import { ERRORS } from '../utils/cell-address.js';


/**

 * Directed graph tracking cell dependencies for efficient recalculation.

 * Uses adjacency list representation: edges go FROM dependency TO dependent.

 */

export class DependencyGraph {

  constructor() {

    // Map from cell address to set of cells that depend on it (forward edges)

    this.edges = new Map();



    // Map from cell address to Cell object (shared reference with Spreadsheet)

    this.nodes = new Map();



    // Cache for topological order to avoid recomputation

    this.cachedOrder = null;

  }



  /**

   * Add a dependency relationship: dependentCell depends on dependencyCell

   * @param {string} dependentAddress - Address of cell containing the formula

   * @param {string} dependencyAddress - Address of cell referenced in formula

   * @throws {Error} If circular reference would be created

   */

  addDependency(dependentAddress, dependencyAddress) {

    // TODO 1: Validate that both addresses are valid cell addresses

    // TODO 2: Check if this dependency already exists (if so, return early)

    // TODO 3: Call detectCircularReference to see if adding this edge would create a cycle

    // TODO 4: If cycle would be created, throw error with ERRORS.CIRCULAR_REF message

    // TODO 5: Get or create the set of dependents for the dependencyAddress

    // TODO 6: Add dependentAddress to the set

    // TODO 7: Invalidate the cached topological order (set this.cachedOrder = null)
```

```
    // TODO 8: Ensure both addresses exist in this.nodes map (create empty cells if needed)
  }

  /**
   * Remove a dependency relationship
   * @param {string} dependentAddress - Address of dependent cell
   * @param {string} dependencyAddress - Address of dependency cell
   */
  removeDependency(dependentAddress, dependencyAddress) {
    // TODO 1: Get the set of dependents for dependencyAddress
    // TODO 2: If set exists, remove dependentAddress from it
    // TODO 3: If set becomes empty after removal, consider deleting it from edges map
    // TODO 4: Invalidate the cached topological order
  }

  /**
   * Get all cells that directly depend on the given cell
   * @param {string} cellAddress - Address of the cell
   * @returns {Set<string>} Set of cell addresses that depend on this cell
   */
  getDependents(cellAddress) {
    // TODO 1: Return the set from this.edges.get(cellAddress) or empty Set if not found
    // TODO 2: Return a copy to prevent external modification (new Set(existing))
  }

  /**
   * Detect if adding an edge would create a circular reference
   * Performs DFS from targetAddress to see if startAddress is reachable
   * @param {string} startAddress - The dependent cell (where edge starts)
   * @param {string} targetAddress - The dependency cell (where edge points)
   * @returns {boolean} True if adding edge would create a cycle
   */
```

```
detectCircularReference(startAddress, targetAddress) {
  // TODO 1: If startAddress equals targetAddress, return true (self-reference creates cycle)
  // TODO 2: Initialize a stack with targetAddress
  // TODO 3: Initialize a visited Set with targetAddress
  // TODO 4: While stack is not empty:
  //   - Pop address from stack
  //   - Get its dependents via getDependents
  //   - For each dependent:
  //     - If dependent equals startAddress, return true (cycle found!)
  //     - If not visited, add to visited and push to stack
  // TODO 5: If loop completes without finding startAddress, return false (no cycle)
}

/**
 * Clear all dependencies for a cell (both incoming and outgoing)
 * Called when cell content is cleared or replaced with non-formula
 * @param {string} cellAddress - Address of cell to clear
 */
clearDependenciesForCell(cellAddress) {
  // TODO 1: Remove this cell from all dependency sets (it's no longer a dependent)
  //   - Iterate through all entries in this.edges
  //   - For each [depAddress, dependentsSet], remove cellAddress from dependentsSet
  // TODO 2: Remove this cell's own dependency set (it no longer depends on others)
  //   - Delete this.edges.get(cellAddress)
  // TODO 3: Invalidate cached topological order
}

/**
 * Get topological order for recalculation (dependencies before dependents)
 * Uses Kahn's algorithm on the subgraph affected by changedCell
 * @param {string} changedCell - Address of the cell that changed
 * @returns {string[]} Array of cell addresses in correct evaluation order
```

```
 */
getTopologicalOrder(changedCell) {
  // TODO 1: Compute affected set: all cells reachable from changedCell via dependency edges
  //   - Perform BFS/DFS from changedCell, following edges forward
  //   - Include changedCell itself in affected set
  // TODO 2: Build adjacency list for affected subgraph only (for efficiency)
  // TODO 3: Compute in-degree for each cell in affected set
  //   - inDegree[cell] = number of dependencies that are also in affected set
  // TODO 4: Initialize queue with cells having in-degree 0
  // TODO 5: Initialize result array for topological order
  // TODO 6: While queue is not empty:
  //   - Dequeue a cell, add to result
  //   - For each dependent of this cell (that's in affected set):
  //     - Decrement dependent's in-degree
  //     - If in-degree becomes 0, enqueue dependent
  // TODO 7: If result length < affected set size, cycle exists (shouldn't happen if we detected earlier)
  // TODO 8: Cache the result in this.cachedOrder for potential reuse
  // TODO 9: Return result array
}

/**
 * Update cell value and trigger recalculation of dependents
 * This is the main entry point for the recalculation engine
 * @param {string} cellAddress - Address of cell that changed
 * @param {any} newValue - New value for the cell
 * @param {Function} evaluator - Function to evaluate formula cells
 */
recalculateFromCell(cellAddress, newValue, evaluator) {
  // TODO 1: Update the cell's value in this.nodes.get(cellAddress)
  // TODO 2: Get topological order for affected cells (call getTopologicalOrder)
  // TODO 3: For each cell in topological order (skipping the changed cell if it's not a formula):
  //   - Get the cell object
```

```
    //   - If cell has a formula, evaluate it using the evaluator function
    //   - If new value differs from old value, mark cell as changed
    // TODO 4: Collect all cells whose values actually changed
    // TODO 5: If any cells changed, recursively call recalculateFromCell for each
    //   (but careful: need to batch to avoid infinite recursion!)
    // TODO 6: Return set of all cells that were updated (for UI refresh)
  }
}
```

Language-Specific Hints:

- **JavaScript Sets and Maps**: Use `new Set()` for dependency sets to ensure uniqueness and O(1) lookups. Use `new Map()` for adjacency lists for clean key-value pairing.
- **Cycle Detection**: Implement DFS iteratively with a stack instead of recursion to avoid stack overflow for long dependency chains: `const stack = [start]; while (stack.length > 0) { ... }`.
- **Topological Sort Optimization**: Compute in-degrees using a `Map` rather than an object for type safety: `const inDegree = new Map(); affectedCells.forEach(cell => inDegree.set(cell, 0));`.
- **Event Emission**: Consider using an event emitter pattern to notify the UI when cells change: `this.emit('cellUpdated', { address, newValue });`.
- **Lazy Evaluation**: Store computed values in the `Cell` objects to avoid re-evaluating formulas unnecessarily. Add a `cachedValue` field that's cleared when dependencies change.

Milestone Checkpoint:

After implementing the `DependencyGraph`, verify correct behavior with these tests:

1. **Basic Dependency Test**:

```javascript
const graph = new DependencyGraph();
graph.addDependency('B1', 'A1'); // B1 = A1+5
graph.addDependency('C1', 'B1'); // C1 = B1*2
const dependents = graph.getDependents('A1');
// Should contain both B1 and C1 (transitive closure)
```

2. **Circular Reference Detection Test**:

```javascript
graph.addDependency('B1', 'A1');

try {
  graph.addDependency('A1', 'B1'); // Should throw circular reference error
  console.error('FAIL: Should have thrown error');
} catch (e) {
  console.log('PASS: Circular reference correctly detected');
}
```

3. **Topological Order Test**:

```javascript
// A1 -> B1 -> C1, and A1 -> D1

graph.addDependency('B1', 'A1');

graph.addDependency('C1', 'B1');

graph.addDependency('D1', 'A1');

const order = graph.getTopologicalOrder('A1');

// Valid orders: [A1, B1, D1, C1] or [A1, D1, B1, C1]

// Must have A1 before B1, B1 before C1, A1 before D1
```

Run these tests manually or integrate them into a test framework. The spreadsheet should now correctly update formulas when their dependencies change, and display `#CIRCULAR!` when users create circular references.

Debugging Tips:

Symptom	Likely Cause	How to Diagnose	Fix
Cell doesn't update when dependency changes	Missing or incorrect dependency edge	Log `getDependents(dependencyCell)` to see if dependent is listed	Ensure `addDependency` is called when formula is parsed
Infinite recalculation loop	Undetected circular reference	Add console.log to `detectCircularReference`; check for indirect cycles	Ensure DFS traverses entire dependency chain, not just immediate dependents
Wrong calculation order	Incorrect topological sort	Print the order array and verify dependencies come before dependents	Check in-degree calculation; ensure you're using forward edges (dependents) not backward
"Maximum call stack size exceeded"	Recursive cycle detection on large graph	Use iterative DFS with explicit stack instead of recursion	Replace recursion with `while (stack.length > 0)` loop
Performance slows with many formulas	Recalculating entire spreadsheet	Count how many cells are evaluated on a single change	Implement true incremental recalculation using affected subgraph only
Dependencies persist after formula change	Not clearing old dependencies	Compare old vs new dependency sets when formula changes	Call `removeDependency` for dependencies no longer referenced

To visualize the dependency graph for debugging, implement a simple text-based graph dump:

```javascript
DependencyGraph.prototype.toString = function() {
  let result = '';
  for (const [dependency, dependents] of this.edges) {
    result += `${dependency} -> ${Array.from(dependents).join(', ')}\n`;
  }
  return result;
};
```

Component Design: Advanced Features Manager

> **Milestone(s):** Milestone 4 (Advanced Features)

This section covers the implementation of copy/paste with reference adjustment, undo/redo using command pattern, CSV import/export, and cell formatting. While the core spreadsheet engine handles reactive calculations, these advanced features transform the application from a basic calculator into a practical tool. They introduce state management challenges, require sophisticated reference transformation logic, and demand careful handling of user expectations around data manipulation.

Mental Model: Time Travel for Spreadsheet Actions

Think of the advanced features as **time travel mechanisms** for spreadsheet operations. Each user action becomes a "moment in time" that can be revisited, modified, or duplicated:

1. **Undo/Redo**: Like a **video player's timeline** - you can rewind to previous states or fast-forward back to later ones. Each edit creates a checkpoint, and the command pattern serves as the recording medium.
2. **Copy/Paste**: Imagine a **stencil with adjustable alignment holes** - when you copy a formula and move it to a new location, the references shift according to the new position unless you pin them in place with absolute references (`$`).
3. **CSV Import/Export**: Acts as a **universal translator** between the spreadsheet's internal data structure and a standard file format that other applications can understand.
4. **Cell Formatting**: Similar to **CSS for numbers** - it separates the raw computational value from its visual presentation, allowing the same number to appear as currency, percentage, or scientific notation.

These features operate at the **orchestration layer**, coordinating between the data model (what gets stored), the presentation layer (what gets shown), and the user's intent (what they want to accomplish).

Advanced Features Interfaces

The advanced features manager integrates with existing components through well-defined interfaces. These interfaces abstract the complexity of state management and coordinate between the spreadsheet engine and UI components.

Core Interface: `AdvancedFeaturesManager`

The main coordinator that routes operations to specialized handlers:

Method	Parameters	Returns	Description
`initialize`	`spreadsheet: Spreadsheet`, `grid: VirtualGrid`	`void`	Sets up internal references and attaches event listeners
`handleCopy`	`sourceRange: {startRow, startCol, endRow, endCol}`	`ClipboardData`	Extracts cells from specified range, serializes with metadata
`handlePaste`	`targetAddress: string`, `clipboardData: ClipboardData`	`void`	Pastes cells at target location, adjusting relative references
`undo`	-	`boolean`	Reverts the most recent change, returns success status
`redo`	-	`boolean`	Reapplies the most recent undone change
`exportToCSV`	`options: ExportOptions`	`string`	Converts spreadsheet data to CSV format string
`importFromCSV`	`csvString: string`, `startAddress: string`	`void`	Parses CSV and inserts data starting at specified cell
`applyFormat`	`address: string | Array<string>`, `format: CellFormat`	`void`	Applies formatting to specified cell(s)
`removeFormat`	`address: string | Array<string>`, `formatProperty?: string`	`void`	Removes formatting (or specific properties) from cell(s)

Supporting Data Structures

Structure	Fields	Description
`ClipboardData`	`cells: Array<ClipboardCell>`, `sourceRange: {startRow, startCol, endRow, endCol}`, `timestamp: number`	Encapsulates copied data with origin information
`ClipboardCell`	`value: any`, `formula: string | null`, `format: CellFormat | null`, `originalAddress: string`	Individual cell data with original location
`Command`	`execute(): void`, `undo(): void`, `getDescription(): string`, `timestamp: number`	Abstract command interface for undo/redo
`ExportOptions`	`includeFormulas: boolean`, `includeFormatting: boolean`, `range: {startRow, startCol, endRow, endCol} | null`	Controls CSV export behavior
`FormatPreset`	`name: string`, `format: CellFormat`	Predefined formatting templates (currency, percentage, etc.)

Format Registry Interface

Manages reusable formatting presets:

Method	Parameters	Returns	Description
`registerPreset`	`name: string`, `format: CellFormat`	`void`	Adds a named formatting preset
`getPreset`	`name: string`	`CellFormat | null`	Retrieves preset by name
`applyPreset`	`address: string`, `presetName: string`	`void`	Applies preset formatting to cell
`listPresets`	-	`Array<string>`	Returns all available preset names

Undo/Redo Algorithm with Command Pattern

The undo/redo system uses the **Command Pattern**, which encapsulates each user action as an object containing both the action and its inverse. This approach separates the execution of commands from their history management.

> **Key Insight**: The command pattern transforms actions into first-class objects that can be stored, replayed, and reversed. This is essential for implementing undo/redo without coupling the UI to specific data model operations.

Command Hierarchy

The system maintains several command types, each implementing the `Command` interface:

1. **`SetValueCommand`**: Changes a cell's raw value
2. **`SetFormulaCommand`**: Changes a cell's formula (and dependencies)
3. **`SetFormatCommand`**: Changes a cell's formatting
4. **`MultiCommand`**: Groups multiple commands for batch operations (like paste)

Algorithm: Command Execution and History Management

1. **Initialization**:

- Create empty undo stack (`Array<Command>`) and redo stack (`Array<Command>`)
- Set maximum stack depth (e.g., 100 commands) to prevent memory issues
- Initialize current state pointer to -1 (no commands executed yet)

2. **Executing a New Command**:
    1. Instantiate the appropriate command object with current and new state
    2. Call `command.execute()` to apply the change
    3. Push command onto undo stack
    4. Clear redo stack (new action invalidates redo chain)
    5. Enforce maximum stack depth by removing oldest commands if exceeded
    6. Update UI to reflect the change
3. **Undo Operation**:
    1. Check if undo stack has commands
    2. Pop the most recent command from undo stack
    3. Call `command.undo()` to revert the change
    4. Push command onto redo stack
    5. Trigger any necessary recalculation for affected cells
    6. Update UI
4. **Redo Operation**:
    1. Check if redo stack has commands
    2. Pop the most recent command from redo stack
    3. Call `command.execute()` to reapply the change
    4. Push command onto undo stack
    5. Trigger any necessary recalculation
    6. Update UI
5. **Batch Operations** (Copy/Paste):
    1. Create a `MultiCommand` container
    2. For each cell in the paste operation, create individual commands
    3. Add all individual commands to the `MultiCommand`
    4. Execute the `MultiCommand` as a single atomic operation
    5. The entire batch appears as one undo/redo step

State Transition Table

Current State	Event	Next State	Actions
`IDLE`	User edits cell	`EXECUTING`	Create command, execute, push to undo, clear redo
`EXECUTING`	Command successful	`IDLE`	Update UI, enable undo button
`EXECUTING`	Command fails	`IDLE`	Show error, discard command
`IDLE`	User clicks Undo	`UNDOING`	Pop from undo stack, execute undo, push to redo
`UNDOING`	Undo complete	`IDLE`	Update UI, enable redo button
`IDLE`	User clicks Redo	`REDOING`	Pop from redo stack, execute, push to undo
`REDOING`	Redo complete	`IDLE`	Update UI

Example Walkthrough: Editing Cell B2

Consider a user editing cell B2:

1. **Initial State**: B2 contains value `10`, undo stack empty
2. **User Action**: Changes B2 to `=A1*2`
3. **System Creates**: `SetFormulaCommand` capturing old value (`10`, null formula) and new value (result of `=A1*2`, formula `=A1*2`)
4. **Execute Command**:
   - Updates cell B2's formula
   - Adds dependency B2→A1 to graph
   - Recalculates B2 (assuming A1=5 → B2=10)
5. **History Update**: Command pushed to undo stack, redo stack cleared
6. **Undo**: Later, user clicks Undo → command pops from undo, `undo()` called:
   - Restores B2's old value `10`
   - Removes dependency B2→A1 from graph
   - Recalculations if needed
   - Command moved to redo stack

Relative/Absolute Reference Adjustment

When copying formulas between cells, relative references must adjust based on the displacement between source and destination. Absolute references (with `$`) remain fixed.

Mental Model: Coordinate System Transformation

Imagine cell references as **GPS coordinates**:

- **Relative references** (`A1`): "5 miles north of my current position"
- **Absolute references** (`$A$1`): "Latitude 40.7128°, Longitude -74.0060°"
- **Mixed references** (`$A1`, `A$1`): Partially anchored

When you move a formula, relative references shift with you, while absolute references stay locked to their original coordinates.

Algorithm: Reference Adjustment

For each formula being copied from source cell `S` to destination cell `D`:

1. **Calculate Displacement**:
   - `rowDelta = D.row - S.row`
   - `colDelta = D.col - S.col`
2. **Parse Source Formula**:
   - Use `parseFormula()` to get AST with `CellReferenceNode` elements
   - Each reference node contains `ParsedCellReference` with `rowAbsolute` and `columnAbsolute` flags
3. **Transform Each Reference**:

```
For each CellReferenceNode in AST:
  If not rowAbsolute:
    newRow = originalRow + rowDelta
    Ensure newRow is within 0..MAX_ROWS-1
  Else:
    newRow = originalRow

  If not columnAbsolute:
    newCol = originalCol + colDelta
    Ensure newCol is within 0..MAX_COLUMNS-1
  Else:
    newCol = originalCol

  Create new address using toCellAddress(newRow, newCol, rowAbsolute, columnAbsolute)
```

4. **Rebuild Formula String**:
   - Traverse transformed AST to regenerate formula text
   - Preserve original spacing and formatting where possible
   - Handle special cases like ranges (adjust both start and end references)
5. **Validation**:
   - Check that adjusted references remain within spreadsheet bounds
   - Detect if adjustment creates circular references
   - Validate syntax of transformed formula

Example: Copying Formula with Mixed References

Source Cell: C5 containing formula `=$A$1 + B2 - C$3 + $D4`

Displacement: Copying to F8 → rowDelta = 3, colDelta = 3

Reference	Type	Original	Adjusted	Reason
`$A$1`	Absolute	Row 0, Col 0	`$A$1`	Both row and column absolute
`B2`	Relative	Row 1, Col 1	`E5`	Row 1+3=4 → E, Col 1+3=4 → 5 (1-based: row 5)
`C$3`	Mixed	Row 2, Col 2	`F$3`	Column relative: 2+3=5 → F, Row absolute: stays 3
`$D4`	Mixed	Row 3, Col 3	`$D7`	Column absolute: stays D, Row relative: 3+3=6 → 7 (1-based)

Result: `=$A$1 + E5 - F$3 + $D7`

Range Adjustment

For range references like `SUM(A1:C5)`:

- Adjust both start and end references independently
- Maintain the range relationship (end must remain ≥ start in both dimensions)
- If adjustment would invert range (end before start), expand to single cell or show error

Architecture Decision Records for Advanced Features

Decision: Command Pattern vs. Memento Pattern for Undo/Redo

Option	Pros	Cons	Chosen?
Command Pattern	Fine-grained control over undo logic; Supports composite commands; Efficient for repetitive operations; Easy to implement selective undo	More classes to maintain; Each operation needs explicit inverse logic	**Yes**
Memento Pattern	Simple implementation; Captures complete state; Minimal logic for new features	Memory intensive (stores full state); Hard to optimize; Doesn't scale to large spreadsheets	No
Operation Logging	Minimal runtime overhead; Simple serialization for persistence	Complex recovery logic; Hard to handle non-commutative operations	No

Decision: Use Command Pattern for Undo/Redo

- **Context**: Need efficient undo/redo that works with large spreadsheets while maintaining performance. Must support complex operations like paste with reference adjustment.
- **Options Considered**: Command pattern, Memento pattern (full state snapshot), Operation logging (replay log).
- **Decision**: Implement command pattern with explicit command objects for each operation type.
- **Rationale**: Command pattern provides optimal memory usage (stores only changes, not full state), enables composite commands for batch operations, and allows fine control over undo logic for complex operations like formula adjustment. The pattern naturally supports unlimited undo depth with configurable limits.
- **Consequences**: Requires more initial code (command classes), but provides better performance and flexibility. Enables future features like macro recording and scriptable operations.

Decision: In-Place vs. Out-of-Place Reference Adjustment

Option	Pros	Cons	Chosen?
In-Place Adjustment	Simple implementation; Direct string manipulation; Fast for simple cases	Error-prone with nested parentheses; Doesn't handle operator precedence; Fails with complex formulas	No
AST-Based Adjustment	Robust handling of all formula types; Preserves formula structure; Easy to validate	Requires full parser; More complex implementation; Slightly slower	**Yes**

Decision: Use AST-Based Reference Adjustment

- **Context**: Need reliable reference adjustment that works correctly with all formula syntax, including functions, ranges, and nested expressions.
- **Options Considered**: String manipulation (regex-based), Abstract Syntax Tree transformation.
- **Decision**: Parse formulas to AST, transform reference nodes in the tree, then regenerate formula string.
- **Rationale**: AST transformation guarantees correct handling of all edge cases: formulas with parentheses, function arguments, mixed references in ranges, and nested expressions. It's the only approach that can't be broken by unusual formula syntax.
- **Consequences**: Requires integrating with the formula parser component. Slightly more overhead than string manipulation, but correctness is critical for user trust.

Decision: CSV as Primary Import/Export Format

Option	Pros	Cons	Chosen?
CSV	Universal compatibility; Simple implementation; Human readable; Standardized	Loses formatting; Limited data types; No formula support	**Yes**
JSON	Rich data representation; Preserves formulas and formatting; Extensible	Less universal; Larger file size; Requires custom parsers	No
Excel Format	Full fidelity; Preserves everything; Professional standard	Extremely complex; Large dependencies; Licensing issues	No
Custom Binary	Compact; Fast parsing; Can preserve everything	No interoperability; Versioning challenges; Debugging difficult	No

Decision: Use CSV for Import/Export

- **Context**: Need a simple, universal file format for data exchange with other applications. Primary use case is sharing data, not preserving full spreadsheet state.
- **Options Considered**: CSV, JSON, Excel (XLSX), Custom binary format.
- **Decision**: Implement CSV import/export as the primary file format.
- **Rationale**: CSV is the lowest common denominator that works with every spreadsheet application, database tool, and text editor. It focuses the feature on data portability rather than full fidelity. Formulas and formatting can be approximated or documented separately.
- **Consequences**: Formulas are exported as their calculated values only. Formatting is lost. However, this matches user expectations for basic data exchange and keeps implementation simple.

Common Pitfalls: Advanced Features

⚠️ Pitfall: Undo Stack Memory Leak

- **Description**: Storing complete cell states in undo commands instead of just the changes, causing unbounded memory growth.
- **Why It's Wrong**: Each undo entry might store the entire spreadsheet state, causing memory usage to grow linearly with every action. With large spreadsheets, this quickly becomes unsustainable.
- **Fix**: Store only the changed properties (delta) in each command. For `SetValueCommand`, store only `{address, oldValue, newValue}` not the entire cell.

⚠️ Pitfall: Incorrect Reference Adjustment with Nested Functions

- **Description**: Using simple string replacement (`B2` → `C3`) that breaks formulas like `SUM(B2, INDEX(B2:B10, 3))`.
- **Why It's Wrong**: String replacement can't distinguish between cell references and other text. It might replace `B2` inside a function name or string literal.
- **Fix**: Always use AST-based adjustment. Parse the formula, identify `CellReferenceNode` elements in the syntax tree, adjust only those, then regenerate the formula string.

⚠️ Pitfall: Forgetting to Clear Redo Stack

- **Description**: Not clearing the redo stack when a new action is performed after undo operations.
- **Why It's Wrong**: If user undoes 3 actions, then performs a new action, the 3 undone actions in the redo stack are no longer reachable but suggest they could be redone. This violates the principle of predictable undo/redo.
- **Fix**: Clear the redo stack whenever a new command is executed (not during undo/redo operations themselves).

⚠️ Pitfall: CSV Export with Commas in Cell Values

- **Description**: Exporting cell value `"Hello, World"` as CSV without proper quoting, breaking the CSV structure.
- **Why It's Wrong**: Unquoted commas in cell values are interpreted as column separators, corrupting the data layout.
- **Fix**: Implement proper CSV encoding: wrap values containing commas, quotes, or newlines in double quotes, and escape embedded quotes by doubling them (`"` → `""`).

⚠️ Pitfall: Formatting Applied to Wrong Cells After Paste

- **Description**: When pasting a range of cells, applying formatting from source to wrong destination cells due to incorrect offset calculation.
- **Why It's Wrong**: If the paste target isn't aligned with the source grid layout, formatting might shift unexpectedly.
- **Fix**: For each cell in the source range, calculate its corresponding destination cell using the displacement vector, not by sequential indexing.

⚠️ Pitfall: Circular References Introduced by Reference Adjustment

- **Description**: Copying a formula that creates a circular reference after adjustment, but not detecting it.
- **Why It's Wrong**: Example: Copying `=A1+1` from B1 to A1 creates `=A1+1` in A1 (circular). The system should detect and prevent this.
- **Fix**: After adjusting references, check if the new formula would create a circular reference using `detectCircularReference()` before applying it.

Implementation Guidance: Advanced Features

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
Undo/Redo	Command pattern with array-based stacks	Command pattern with persistent storage (IndexedDB) for unlimited history
Clipboard	Browser Clipboard API with custom formats	Custom clipboard manager with compression and format negotiation
CSV Handling	Simple string concatenation with regex parsing	Streaming parser/generator for large datasets
Formatting	Inline styles on cell elements	CSS classes with style sheet generation

B. Recommended File/Module Structure

```
spreadsheet-app/
├── src/
│   ├── advanced-features/
│   │   ├── commands/
│   │   │   ├── Command.js           # Abstract base command
│   │   │   ├── SetValueCommand.js
│   │   │   ├── SetFormulaCommand.js
│   │   │   ├── SetFormatCommand.js
│   │   │   └── MultiCommand.js
│   │   ├── clipboard/
│   │   │   ├── ClipboardManager.js
│   │   │   ├── ReferenceAdjuster.js
│   │   │   └── clipboard-utils.js
│   │   ├── history/
│   │   │   ├── HistoryManager.js
│   │   │   └── HistoryStack.js
│   │   ├── import-export/
│   │   │   ├── CSVExporter.js
│   │   │   ├── CSVImporter.js
│   │   │   └── csv-utils.js
│   │   ├── formatting/
│   │   │   ├── FormatManager.js
│   │   │   ├── FormatPresetRegistry.js
│   │   │   └── style-mapping.js
│   │   └── AdvancedFeaturesManager.js  # Main coordinator
│   ├── core/          # Existing core components
│   └── ui/            # Existing UI components
```

C. Infrastructure Starter Code

Complete History Stack Implementation:

JAVASCRIPT

```javascript
// src/advanced-features/history/HistoryStack.js

export class HistoryStack {

  constructor(maxDepth = 100) {

    this.maxDepth = maxDepth;

    this.undoStack = [];

    this.redoStack = [];

  }


  push(command) {

    this.undoStack.push(command);



    // Enforce maximum depth

    if (this.undoStack.length > this.maxDepth) {

      this.undoStack.shift(); // Remove oldest

    }



    // Clear redo stack on new action

    this.clearRedo();

  }


  undo() {

    if (this.canUndo()) {

      const command = this.undoStack.pop();

      this.redoStack.push(command);

      return command;

    }

    return null;

  }


  redo() {

    if (this.canRedo()) {

      const command = this.redoStack.pop();

      this.undoStack.push(command);

      return command;
```

```
    }
    return null;
  }

  canUndo() {
    return this.undoStack.length > 0;
  }

  canRedo() {
    return this.redoStack.length > 0;
  }

  clearRedo() {
    this.redoStack = [];
  }

  clearAll() {
    this.undoStack = [];
    this.redoStack = [];
  }

  getUndoDescription() {
    return this.canUndo() ? this.undoStack[this.undoStack.length - 1].getDescription() : null;
  }

  getRedoDescription() {
    return this.canRedo() ? this.redoStack[this.redoStack.length - 1].getDescription() : null;
  }
}
```

Complete CSV Export Utility:

JAVASCRIPT

```javascript
// src/advanced-features/import-export/csv-utils.js

export class CSVUtils {

  static escapeCSVValue(value) {

    if (value === null || value === undefined) {

      return '';

    }



    const stringValue = String(value);



    // Check if value needs quoting

    const needsQuotes = stringValue.includes(',') ||

                        stringValue.includes('"') ||

                        stringValue.includes('\n') ||

                        stringValue.includes('\r');



    if (!needsQuotes) {

      return stringValue;

    }



    // Escape double quotes by doubling them

    const escapedValue = stringValue.replace(/"/g, '""');

    return `"${escapedValue}"`;

  }


  static parseCSVLine(line) {

    const result = [];

    let current = '';

    let inQuotes = false;



    for (let i = 0; i < line.length; i++) {

      const char = line[i];

      const nextChar = line[i + 1];
```

```
      if (char === '"' && inQuotes && nextChar === '"') {
        // Escaped quote inside quoted field
        current += '"';
        i++; // Skip next quote
      } else if (char === '"') {
        // Quote character
        inQuotes = !inQuotes;
      } else if (char === ',' && !inQuotes) {
        // End of field
        result.push(current);
        current = '';
      } else {
        // Regular character
        current += char;
      }
    }

    // Add last field
    result.push(current);
    return result;
  }
}
```

D. Core Logic Skeleton Code

Abstract Base Command:

```javascript
// src/advanced-features/commands/Command.js

export class Command {
  constructor(description) {
    this.description = description;
    this.timestamp = Date.now();
  }

  // TODO 1: Execute the command, applying changes to the spreadsheet
  // TODO 2: Should return true if successful, false if failed
  execute() {
    throw new Error('Subclass must implement execute()');
  }

  // TODO 3: Revert the changes made by execute()
  // TODO 4: Should restore the spreadsheet to state before execute()
  undo() {
    throw new Error('Subclass must implement undo()');
  }

  // TODO 5: Return human-readable description for UI display
  getDescription() {
    return this.description;
  }
}
```

SetFormulaCommand Implementation Skeleton:

```javascript
// src/advanced-features/commands/SetFormulaCommand.js

import { Command } from './Command.js';

export class SetFormulaCommand extends Command {
  constructor(cellAddress, oldFormula, newFormula, oldValue, spreadsheet) {
    // TODO 1: Call parent constructor with descriptive string
    // TODO 2: Store all parameters as instance properties
    // TODO 3: Store old and new dependencies if needed for graph updates
  }

  execute() {
    // TODO 4: Validate that newFormula is different from oldFormula
    // TODO 5: Parse the new formula to get dependencies
    // TODO 6: Check for circular references before applying
    // TODO 7: Update the cell's formula in the spreadsheet data model
    // TODO 8: Update dependency graph (remove old deps, add new deps)
    // TODO 9: Trigger recalculation starting from this cell
    // TODO 10: Return true if successful
  }

  undo() {
    // TODO 11: Restore the old formula to the cell
    // TODO 12: Update dependency graph (remove new deps, restore old deps)
    // TODO 13: Restore the old value if it was cached
    // TODO 14: Trigger recalculation if needed
    // TODO 15: Return true if successful
  }
}
```

JAVASCRIPT

ReferenceAdjuster Core Logic:

```javascript
// src/advanced-features/clipboard/ReferenceAdjuster.js

export class ReferenceAdjuster {

  static adjustFormula(formula, sourceAddress, targetAddress) {

    // TODO 1: Parse source and target addresses to get row/col indices

    // TODO 2: Calculate rowDelta = targetRow - sourceRow

    // TODO 3: Calculate colDelta = targetCol - sourceCol

    // TODO 4: Parse the formula string into an AST using parseFormula()

    // TODO 5: Traverse AST and find all CellReferenceNode and RangeNode

    // TODO 6: For each reference node, check rowAbsolute and columnAbsolute flags

    // TODO 7: Apply delta to non-absolute rows/columns

    // TODO 8: Validate adjusted references are within spreadsheet bounds

    // TODO 9: For RangeNode, adjust both start and end references

    // TODO 10: Regenerate formula string from modified AST

    // TODO 11: Return adjusted formula string

  }


  static adjustRange(startAddress, endAddress, sourceAddress, targetAddress) {

    // TODO 12: Parse all addresses to row/col indices

    // TODO 13: Calculate displacement between source and target

    // TODO 14: Adjust start address using adjustSingleReference()

    // TODO 15: Adjust end address using adjustSingleReference()

    // TODO 16: Ensure adjusted range is valid (end >= start)

    // TODO 17: Return adjusted range string "start:end"

  }

}
```

FormatManager Apply Format Skeleton:

```javascript
// src/advanced-features/formatting/FormatManager.js

export class FormatManager {

  applyFormat(address, format, spreadsheet, grid) {

    // TODO 1: Parse address (could be single cell "A1" or range "A1:B10")

    // TODO 2: If range, expand to individual cell addresses

    // TODO 3: For each cell address:

    // TODO 4:   Get the cell from spreadsheet

    // TODO 5:   Create or update cell.format object

    // TODO 6:   Apply format properties (merge with existing)

    // TODO 7:   Update cell display via grid.updateCellStyle()

    // TODO 8: Create and return a SetFormatCommand for undo/redo

  }


  createFormatPreset(name, baseFormat) {

    // TODO 9: Validate preset name doesn't exist

    // TODO 10: Create enhanced format with defaults filled

    // TODO 11: Store in preset registry

    // TODO 12: Return the complete preset object

  }

}
```

E. Language-Specific Hints

1. **JavaScript Clipboard API**: Use `navigator.clipboard.writeText()` for CSV export and `navigator.clipboard.readText()` for import. For richer clipboard data, use the newer `navigator.clipboard.write()` with custom MIME types.
2. **Event Delegation for Formatting**: Attach a single click handler to the grid container for format painting, using event delegation to identify which cell was clicked.
3. **Immutable Updates for Undo**: Consider using immutable data patterns for spreadsheet state to simplify undo/redo. Libraries like Immer can help but add dependency.
4. **Web Workers for CSV Processing**: For large spreadsheets, use Web Workers to parse/generate CSV in the background without blocking the UI thread.
5. **CSS Custom Properties for Themes**: Use CSS variables for formatting presets to allow theme switching without JavaScript changes:

```css
:root {
  --currency-color: #2e7d32;
  --percentage-color: #1565c0;
  --error-color: #c62828;
}
```

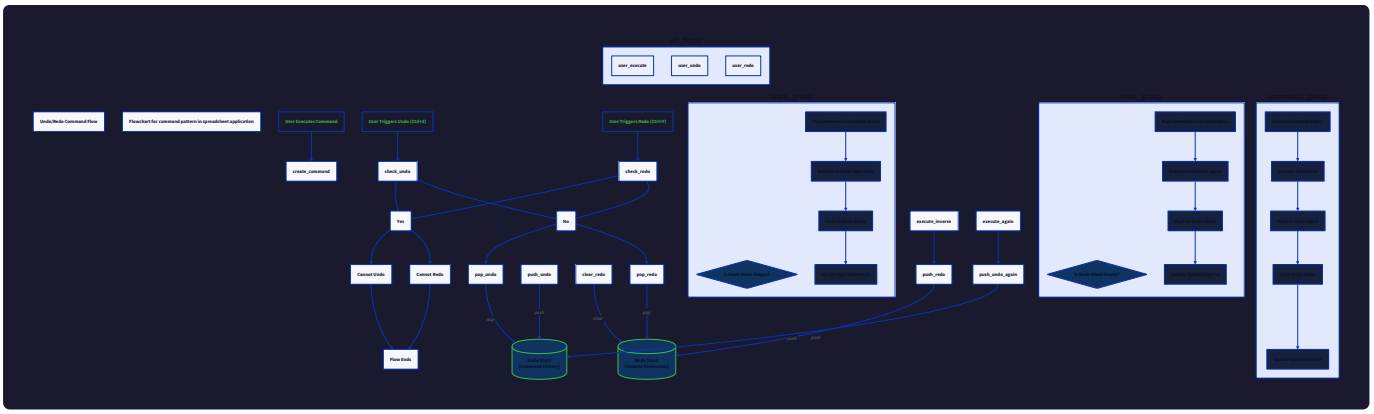
F. Milestone Checkpoint

After implementing advanced features:

1. **Test Undo/Redo**:
   - Edit a cell value, then press Ctrl+Z (or click Undo)
   - Value should revert to previous state
   - Press Ctrl+Y (or click Redo) to reapply
   - **Expected**: Smooth transitions with no errors in console
2. **Test Copy/Paste**:
   - Enter formula `=A1+B2` in cell C3
   - Copy C3 and paste to D4
   - **Expected**: D4 shows formula `=B2+C3` (references adjusted by +1 row, +1 col)
3. **Test CSV Export**:
   - Create data in cells A1:C3
   - Click "Export CSV"
   - Open downloaded file in text editor
   - **Expected**: Properly formatted CSV with commas between values
4. **Test Formatting**:
   - Select a cell with number 0.25
   - Apply "Percentage" format
   - **Expected**: Cell displays "25%" but value remains 0.25

G. Debugging Tips

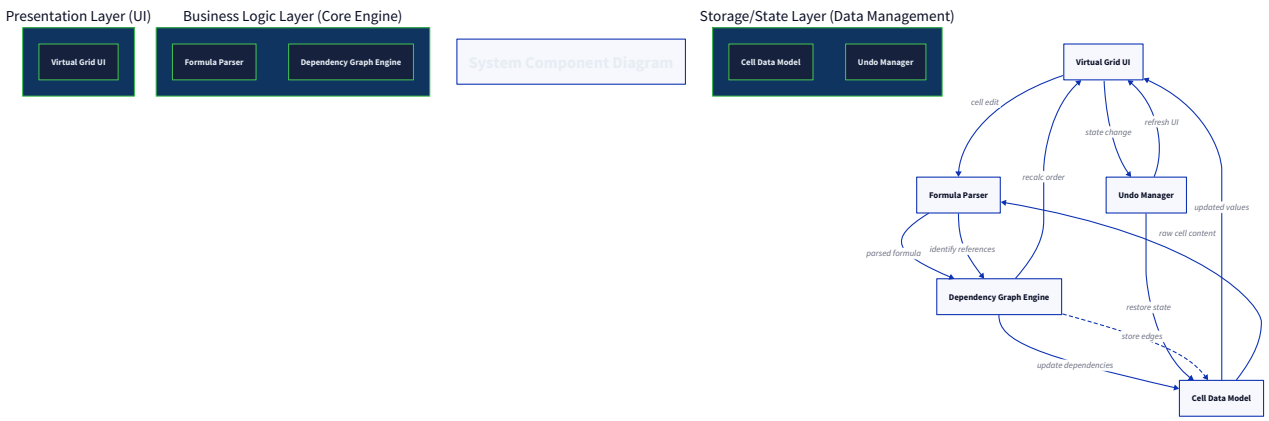
Symptom	Likely Cause	How to Diagnose	Fix
Undo does nothing	Command not properly stored	Check `HistoryStack.push()` is called after each action	Ensure command execution triggers history recording
Pasted formulas show `#REF!`	Reference adjustment out of bounds	Log original and adjusted addresses	Add bounds checking in `ReferenceAdjuster`
CSV import creates wrong columns	Commas in cell values not escaped	Export test data and examine in text editor	Implement proper CSV escaping in `CSVUtils.escapeCSVValue()`
Formatting lost on undo	Format not captured in command	Check `SetFormatCommand` stores old format	Ensure command captures complete cell format state
Copy/paste of range misaligns	Incorrect displacement calculation	Log source range and target address during paste	Verify `rowDelta` and `colDelta` calculation
Redo stack not clearing	New action after undo not clearing redo	Trace command execution flow	Call `history.clearRedo()` in `execute()` method
Performance slow with many formats	Inline styles on each cell	Check browser DevTools for many style rules	Use CSS classes and update class list instead



Interactions and Data Flow

> **Milestone(s):** Milestone 1 (Grid & Cell Rendering), Milestone 2 (Formula Parser), Milestone 3 (Dependency Graph & Recalculation), Milestone 4 (Advanced Features)

This section traces the complete journey of user actions through the system—from a simple keystroke to a complex chain of recalculations—explaining how components collaborate to maintain consistency. Understanding these data flows is critical for debugging and extending the spreadsheet. We'll examine three core interaction patterns: editing a cell with a formula, copying and pasting with reference adjustment, and managing undo/redo state.



Cell Edit and Recalculation Flow

Mental Model: Ripples in a Computational Pond

Imagine dropping a pebble (changing a cell) into a pond. The initial splash is the cell edit, but the ripples that spread outward are the dependent cells recalculating in perfect order. The system's job is to ensure ripples never collide (circular references) and always travel in the correct direction (topological order).

This flow represents the spreadsheet's **reactive dataflow** core. When a user enters or modifies a formula, the system must parse it, update dependency relationships, and recalculate all affected cells while preserving consistency.

Detailed Interaction Sequence

The complete sequence involves six stages: UI capture, formula parsing, graph update, topological ordering, evaluation, and UI refresh.

1. **UI Capture Phase**: The `VirtualGrid` detects user input.
2. **Parsing Phase**: The `FormulaParser` transforms raw text into an executable structure.
3. **Graph Update Phase**: The `DependencyGraph` modifies dependency edges and checks for cycles.
4. **Ordering Phase**: The system determines the correct recalculation sequence.
5. **Evaluation Phase**: Each affected cell's formula is evaluated with current values.
6. **UI Refresh Phase**: Changed cell values are rendered in the grid.

Component Interactions Table

Component	Responsibility in This Flow	Key Methods Invoked
`VirtualGrid`	Captures edit event, manages edit input, triggers recalculation	`enterEditMode()`, `setSelection()`, `renderViewport()`
`Spreadsheet`	Coordinates the overall process, updates cell data	`setCellFormula()`, `getCell()`, `getCellByAddress()`
`FormulaParser`	Parses formula string to AST and extracts dependencies	`parseFormula()`, `extractDependencies()`
`DependencyGraph`	Updates dependency relationships, detects cycles	`addDependency()`, `removeDependency()`, `detectCircularReference()`
Recalculation Engine	Orders and executes cell evaluations	`getTopologicalOrder()`, `evaluateAST()`

Step-by-Step Walkthrough: Editing Cell B2 to "=A1*2"

Let's trace a concrete example where cell B2 (initially empty) receives the formula `=A1*2`, while A1 contains the value `10`.

1. **User Action**: User double-clicks cell B2 (row 1, column 1 in zero-based indexing), types `=A1*2`, and presses Enter.
2. **Grid Event Handling**: The `VirtualGrid`'s edit input blur handler calls `grid.handleCellEdit(1, 1, "=A1*2")`.
3. **Spreadsheet Update**: `Spreadsheet.setCellFormula(1, 1, "=A1*2")` creates a `Cell` with `formula` set and `value` temporarily cleared.
4. **Formula Parsing**: `FormulaParser.parseFormula("=A1*2", "B2")` returns a `ParsedFormula` with:
   - `ast`: A `BinaryOpNode` with operator `*`, left as `CellReferenceNode("A1")`, right as `NumberNode(2)`
   - `dependencies`: `Set(["A1"])`
   - `hasErrors`: `false`
5. **Dependency Graph Update**:
   - The system removes B2's old dependencies (empty set initially)
   - For each new dependency (A1), calls `DependencyGraph.addDependency("B2", "A1")`
   - Cycle detection runs: `detectCircularReference("A1", "B2")` returns `false` (no cycle)
6. **Dirty Cell Collection**: The system marks B2 as "dirty" (needing recalculation).
7. **Topological Ordering**: `DependencyGraph.getTopologicalOrder("B2")` returns `["A1", "B2"]` (dependencies before dependents).
8. **Cell Evaluation Loop**:
   - **A1**: Not dirty (value remains `10`)
   - **B2**: `evaluateAST(binaryOpNode, valueResolver)` resolves A1 to `10`, computes `10*2`, stores `20` as B2's `value`
9. **UI Refresh**: `VirtualGrid.renderViewport()` updates the visual representation of B2 to display `20`.
10. **History Recording**: `AdvancedFeaturesManager` pushes a `Command` to the undo stack capturing B2's previous state (empty) and new state (formula `=A1*2`, value `20`).

State Transition During Edit

Current State	Event	Next State	Actions Taken
Steady State	User begins editing B2	Edit Mode	`VirtualGrid.enterEditMode()` shows input, saves previous value
Edit Mode	User submits formula "=A1*2"	Parsing	`Spreadsheet.setCellFormula()` updates cell, triggers parse
Parsing	Parse completes successfully	Graph Update	Dependencies extracted, old edges removed, new edges added
Graph Update	No circular reference detected	Ordering	Dirty cells collected, topological sort performed
Ordering	Order ["A1", "B2"] determined	Evaluating	Cells evaluated in sequence with current values
Evaluating	B2 evaluation produces 20	Updating	Cell values updated, change notifications sent
Updating	Values committed	Steady State	UI refreshed, history recorded

Error Scenario: Circular Reference If the user attempts to set A1 to `=B2+1` (creating A1→B2→A1), the flow diverges at step 5:

- `detectCircularReference("B2", "A1")` traverses dependencies and finds A1 already in B2's dependency chain.
- The method returns `true`, preventing the edge addition.
- Cell A1's `value` is set to `ERRORS.CIRCULAR_REF`, `hasErrors` becomes `true`.
- The UI displays `#CIRCULAR!` in cell A1.
- No recalculation occurs; the graph remains unchanged.

Architecture Decision: Immediate vs. Deferred Recalculation

> **Decision: Immediate Recalculation on Every Change**
>
> - **Context**: After any cell edit, dependent cells must reflect new values. We must choose between recalculating immediately or queuing changes.
> - **Options Considered**:
>   1. **Immediate Recalculation**: Synchronously recalc after each edit before returning control to UI.
>   2. **Debounced Recalculation**: Batch multiple rapid edits (within 100ms) into a single recalculation.
>   3. **Manual Recalculation**: Require explicit user action (like pressing F9) to recalc.
> - **Decision**: Immediate recalculation for simplicity and predictable feedback.
> - **Rationale**: Educational projects benefit from straightforward cause-effect visibility. Immediate feedback helps users understand dependency chains. Complexity of batching (tracking dirty cells across multiple edits) outweighs performance benefits for grids under 10,000 cells.
> - **Consequences**: The UI may feel slightly sluggish during large recalc chains, but the system remains deterministic and easier to debug.

Option	Pros	Cons	Chosen?
Immediate	Simple implementation, predictable	May cause UI lag with long dependency chains	**Yes**
Debounced	Better performance for rapid typing	Complex state tracking, timing issues	No
Manual	Maximum performance, explicit control	Poor user experience, un-Excel-like	No

Common Pitfalls in Cell Edit Flow

⚠️ Pitfall: Forgetting to Clear Old Dependencies
When changing a formula from `=A1+B1` to `=C1*2`, failing to remove the A1 and B1 dependencies before adding C1 creates "phantom dependencies" where B2 incorrectly recalculates when A1 changes.
Why it's wrong: The dependency graph becomes inconsistent with actual formulas, causing unnecessary recalculations or missing required ones.
Fix: Always call `removeDependency()` for each old dependency before adding new ones.

⚠️ Pitfall: Not Handling Parse Errors Gracefully
If `parseFormula()` throws an exception on malformed input like `=A1++2`, the entire edit flow crashes.
Why it's wrong: User experience suffers; spreadsheet enters inconsistent state.
Fix: Wrap parsing in try-catch, store error in `ParsedFormula.hasErrors`, display error value in cell (e.g., `#PARSE!`), and skip dependency updates.

⚠️ Pitfall: Recursive Recalculation Trigger
Calling `renderViewport()` during recalculation might read cell values that are mid-update, causing visual glitches.
Why it's wrong: Race condition between evaluation and rendering produces flickering or incorrect displays.
Fix: Complete all value updates before any UI refresh. Use a batch update flag or separate mutation phase.

Copy/Paste with Reference Adjustment Flow

Mental Model: Transplanting with Adaptive Connections

Imagine copying a fragment of a living circuit board. Each component (cell) has wires (references) connecting to other components. When you transplant the fragment to a new location, relative wires must stretch to point to new positions relative to the move, while absolute wires remain fixed. The system must rewire these connections intelligently.

This flow handles the complex logic of duplicating cell content—especially formulas—while adjusting cell references based on relative vs. absolute notation and the displacement between source and target locations.

Key Concepts in Reference Adjustment

- **Relative Reference**: `A1` or `B3` — adjusts based on the **displacement vector** between source and target.
- **Absolute Reference**: `$A$1` or `B$3` — column and/or row fixed with `$`, does not adjust.
- **Mixed Reference**: `$A1` or `A$1` — partially absolute, partially relative.
- **Displacement Vector**: The row and column differences between the top-left of the source range and the top-left of the target range.

Two-Phase Copy/Paste Process

Phase	Description	Key Components
Copy Phase	Extract cells from source range, serialize formulas and metadata	`ClipboardManager`, `VirtualGrid`
Paste Phase	Insert cells at target, adjust references, update dependencies	`AdvancedFeaturesManager`, `DependencyGraph`

Step-by-Step Walkthrough: Copying B2:C3 to E5:F6

Assume we have a 2×2 block:

- B2: `=A1*2` (relative)
- B3: `=SUM($A$1:A2)` (mixed absolute/relative)
- C2: `=B2+1` (relative)
- C3: `=C2*$B$1` (mixed)

1. **Copy Initiation**: User selects range B2:C3 (4 cells), presses Ctrl+C.
2. **Range Extraction**: `AdvancedFeaturesManager.handleCopy({start: "B2", end: "C3"})` creates a `ClipboardData` object:

```javascript
{
  cells: [
    {value: 20, formula: "=A1*2", format: null, originalAddress: "B2"},
    {value: 30, formula: "=SUM($A$1:A2)", format: null, originalAddress: "B3"},
    // ... similar for C2, C3
  ],
  sourceRange: {startRow: 1, startCol: 1, endRow: 2, endCol: 2},
  timestamp: 1678901234567
}
```

3. **Clipboard Storage**: The `ClipboardData` is stored in the `AdvancedFeaturesManager` (and optionally in the system clipboard via `navigator.clipboard`).
4. **Paste Initiation**: User selects cell E5 (row 4, column 4) and presses Ctrl+V.
5. **Displacement Calculation**:
   - Source top-left: (row=1, col=1) → B2
   - Target top-left: (row=4, col=4) → E5
   - Displacement: (Δrow=3, Δcol=3)
6. **Cell-by-Cell Processing**: For each cell in `ClipboardData.cells`:
   - **Original B2** (row=1, col=1) → **Target E5** (row=4, col=4)
   - Formula `=A1*2` becomes `=D4*2` (A1 → D4: column 0→3, row 0→3)
   - **Original B3** (row=2, col=1) → **Target E6** (row=5, col=4)
   - Formula `=SUM($A$1:A2)` becomes `=SUM($A$1:D5)`:
     - `$A$1` remains `$A$1` (absolute)
     - `A2` (row=1, col=0) → `D5` (row=4, col=3): both adjust by Δrow=3, Δcol=3
7. **Graph Update for Each Pasted Cell**:
   - Remove existing dependencies from target cells (if overwriting)
   - Parse adjusted formula to extract new dependencies
   - Add new dependencies to the graph
   - Check for circular references
8. **Recalculation**: Topological sort and evaluation for all affected cells (including dependents of pasted cells).
9. **UI Refresh**: Update the viewport to show new values in E5:F6.

Reference Adjustment Algorithm

The core logic lives in `adjustFormula(formula, sourceAddress, targetAddress)`:

1. Parse the formula into tokens, identifying cell reference tokens.
2. For each cell reference token:
   - Parse the address into `ParsedCellReference` (row, column, rowAbsolute, columnAbsolute)
   - If `columnAbsolute` is false: newColumn = originalColumn + Δcol
   - If `rowAbsolute` is false: newRow = originalRow + Δrow
   - Reconstruct the address string using new indices and absolute flags

3. Reassemble the formula string with adjusted references.
4. Return the adjusted formula.

Edge Cases in Paste Operations

Edge Case	Handling Strategy	Example
Paste over existing formulas	Remove old dependencies before adding new ones	Overwriting C2 with new formula must break C2's old dependency links
Paste with partial absolute references	Only adjust the relative parts	`A$1` pasted 2 rows down becomes `C$1` (column adjusts, row fixed)
Paste beyond grid boundaries	Truncate to valid range, show warning	Pasting 10 columns when only 5 columns remain → paste first 5
Paste values only (without formulas)	Strip formulas, paste only raw values	Use "Paste Special" → Values option

Architecture Decision: In-Memory vs. System Clipboard

> **Decision: Dual Clipboard Strategy with Primary In-Memory Storage**
>
> - **Context**: We need to copy/paste within the application and potentially with other applications.
> - **Options Considered**:
>   1. **System Clipboard Only**: Use `navigator.clipboard.writeText()` with custom serialization format.
>   2. **In-Memory Only**: Store `ClipboardData` in a JavaScript variable, faster but doesn't work with other apps.
>   3. **Dual Strategy**: Try system clipboard, fall back to in-memory; prioritize in-memory for complex operations.
> - **Decision**: Dual strategy with primary in-memory storage for reliability and control.
> - **Rationale**: System clipboard APIs have permission restrictions and asynchronous complexity. In-memory storage guarantees instant access for reference adjustment logic. We can still attempt system clipboard for interoperability with simple text (CSV format).
> - **Consequences**: Copies won't persist across browser refresh (unless we also use system clipboard). Trade-off: better control vs. reduced external interoperability.

Option	Pros	Cons	Chosen?
System Clipboard Only	Works with other apps, persistent	Permission prompts, async complexity, format limitations	No
In-Memory Only	Fast, reliable, full control	Not shared with other apps, lost on refresh	Partial
Dual Strategy	Best of both, fallback options	Implementation complexity, two code paths	**Yes**

Common Pitfalls in Copy/Paste Flow

⚠️ Pitfall: Not Adjusting Range References Correctly
The formula `=SUM(A1:B2)` pasted with displacement (2, 2) should become `=SUM(C3:D4)`, not `=SUM(A1:B2)`.
Why it's wrong: Range endpoints must be adjusted individually; treating the range as a single token breaks relative references.
Fix: Parse ranges into start and end references, adjust each separately, then reconstruct.

⚠️ Pitfall: Forgetting to Update Dependencies for Overwritten Cells
When pasting over cell C2 that previously contained `=A1+B1`, the old dependencies (A1, B1) must be removed from the graph.

Why it's wrong: The graph retains stale dependencies, causing unnecessary recalculations when A1 changes.
Fix: Before setting new cell content, clear all existing dependencies for the target cell.

⚠️ **Pitfall: Infinite Loop in Self-Referential Paste**
Pasting a formula that, after adjustment, refers to itself (e.g., pasting `=A1+1` from B2 to A1 creates `=#REF!+1` if not handled).
Why it's wrong: The adjusted formula creates a circular reference or invalid reference.
Fix: After adjustment, validate that the formula doesn't reference any cell in the paste target range before committing.

Undo/Redo State Management Flow

Mental Model: Time-Travel Tape Recorder
Imagine a cassette tape recording every change to the spreadsheet. The playhead represents the current state. "Undo" rewinds the tape one action, "Redo" fast-forwards one action. Each recording captures not just the new state, but enough information to reverse the action precisely.

This flow implements the **command pattern** to enable reversible operations. Each user action is encapsulated as a `Command` object that knows how to execute and invert itself.

Command Pattern Architecture

Component	Role	Responsibility
`Command`	Abstract command	Defines `execute()` and `undo()` methods
`SetCellValueCommand`	Concrete command	Captures cell address, previous value, new value
`SetCellFormulaCommand`	Concrete command	Captures cell address, previous formula/state, new formula
`HistoryStack`	Command manager	Maintains undo and redo stacks, enforces depth limits
`AdvancedFeaturesManager`	Coordinator	Creates commands, pushes to history, executes undo/redo

Three-Stack State Model

The system maintains three logical stacks:

1. **Undo Stack**: Commands that can be undone (most recent at top)
2. **Redo Stack**: Commands that can be redone (after undo)
3. **Future Stack** (conceptual): New commands after undo replace redo stack

Step-by-Step Walkthrough: Setting B2 to "=A1*2", Then Undo/Redo

1. **Initial State**: B2 is empty. Both undo and redo stacks are empty.
2. **User Action**: Enter formula `=A1*2` into B2.
3. **Command Creation**: `AdvancedFeaturesManager` creates a `SetCellFormulaCommand`:

```javascript
{
  description: "Set formula in B2",
  timestamp: 1678901234567,
  cellAddress: "B2",
  previousState: {value: null, formula: null, format: null},
  newState: {value: 20, formula: "=A1*2", format: null}
}
```

4. **Command Execution**: The command's `execute()` method calls `Spreadsheet.setCellFormula(1, 1, "=A1*2")`, triggering the full recalculation flow.
5. **History Recording**: `HistoryStack.push(command)` adds the command to the undo stack and clears the redo stack.
6. **UI Update**: Grid displays `20` in B2.
7. **Undo Request**: User presses Ctrl+Z.
8. **Undo Execution**: `HistoryStack.undo()` pops the command from undo stack, calls `command.undo()`:
   - `undo()` restores B2 to `previousState` (empty)
   - Dependency graph removes B2→A1 edge
   - Command is pushed to redo stack
9. **UI Update**: Grid shows B2 as empty.
10. **Redo Request**: User presses Ctrl+Y.
11. **Redo Execution**: `HistoryStack.redo()` pops from redo stack, calls `command.execute()` again:
    - Formula is restored, dependencies re-added
    - Command moves back to undo stack
12. **UI Update**: Grid shows `20` in B2 again.

Compound Commands for Batch Operations

Some operations like paste or clear range involve multiple cells. These are wrapped in `CompoundCommand`:

Command Type	Use Case	Example
Atomic Command	Single cell change	Set B2 value
Compound Command	Multi-cell operation	Paste 2×2 block, clear range A1:B10
Macro Command	Sequence of unrelated operations	User macro recording

State Transition Table for Undo/Redo

Current State	Event	Next State	Actions
Clean	User modifies cell	Modified	Create command, execute, push to undo stack, clear redo
Modified	User performs another modification	Modified (new)	Create new command, push to undo, clear redo
Modified	User triggers undo	Undone	Pop command from undo, execute undo(), push to redo
Undone	User triggers undo again	Further undone	Pop next command, undo, push to redo
Undone	User triggers redo	Redone	Pop command from redo, execute(), push to undo
Undone	User makes new modification	Modified	Clear redo stack, create/push new command

Error Recovery in Undo/Redo

Failure Mode	Detection	Recovery Strategy
Undo fails (e.g., dependency graph error)	`command.undo()` throws exception	Roll back partial changes, restore command to undo stack, show error to user
Redo stack inconsistency	Command references deleted cells	Validate command state before execution, skip invalid commands with warning
Memory exhaustion	Undo stack exceeds `maxDepth`	Trim oldest commands from stack, maintain only most recent N

Architecture Decision: Deep vs. Shallow Command State

> **Decision: Deep Copy of Cell State in Commands**
>
> - **Context**: Commands must store enough state to restore cells exactly during undo.
> - **Options Considered**:
>   1. **Shallow Reference**: Store only cell address and new value; rely on current state for undo.
>   2. **Deep Copy**: Store complete previous and new cell state (value, formula, format, dependencies).
>   3. **Delta Encoding**: Store only what changed (e.g., formula changed from X to Y).
> - **Decision**: Deep copy of relevant cell properties (excluding computed dependencies).
> - **Rationale**: While memory-intensive, this approach simplifies undo logic and isolates commands from spreadsheet implementation changes. Dependencies are recomputed during undo/redo execution rather than stored. For educational projects, clarity trumps memory optimization.
> - **Consequences**: Larger memory footprint (especially for multi-cell operations), but robust undo that works with any cell property change.

Option	Pros	Cons	Chosen?
Shallow Reference	Minimal memory, simple	Cannot undo if cell changes again (dangling reference)	No
Deep Copy	Complete isolation, reliable	Memory intensive, requires copying logic	**Yes**
Delta Encoding	Balanced memory/functionality	Complex to implement for all change types	No

Common Pitfalls in Undo/Redo Flow

⚠️ Pitfall: Not Clearing Redo Stack on New Action
After undoing two actions, then making a new edit, the redo stack should be cleared.

Why it's wrong: User expects redo to apply to the current timeline; allowing redo after new action creates confusing state transitions.
Fix: Call `HistoryStack.clearRedo()` whenever a new command is pushed after undo.

⚠️ Pitfall: Forgetting to Recalculate After Undo
Restoring a cell's previous formula must trigger dependency graph updates and recalculation, not just set the value.
Why it's wrong: Dependencies remain inconsistent, dependent cells show stale values.
Fix: Command's `undo()` should call full `setCellFormula()` or `setCellValue()` methods, not directly manipulate cell data.

⚠️ Pitfall: Memory Leak from Unbounded History
Storing every change indefinitely can crash the browser with large spreadsheets.
Why it's wrong: Memory usage grows without bound; performance degrades over time.
Fix: Implement `HistoryStack.maxDepth` (e.g., 100 commands), trim oldest entries when exceeded.

Implementation Guidance

Technology Recommendations

Component	Simple Option	Advanced Option
Undo/Redo Storage	In-memory JavaScript arrays	IndexedDB for persistent history across sessions
Clipboard Integration	In-memory `ClipboardData` object	`Clipboard API` + custom MIME types for Excel interoperability
Event Delegation	Direct event listeners on grid cells	Event delegation on grid container with data attributes
State Management	Direct method calls between components	Event bus (pub/sub) for loose coupling

Recommended File Structure

```
src/
├── interactions/
│   ├── command-pattern/
│   │   ├── Command.js           # Abstract base command
│   │   ├── SetCellValueCommand.js
│   │   ├── SetCellFormulaCommand.js
│   │   ├── CompoundCommand.js   # For multi-cell operations
│   │   └── CommandFactory.js    # Creates commands from actions
│   ├── clipboard/
│   │   ├── ClipboardManager.js
│   │   ├── ReferenceAdjuster.js # adjustFormula() implementation
│   │   └── CSVSerializer.js     # For system clipboard CSV format
│   └── history/
│       ├── HistoryStack.js
│       └── HistoryManager.js    # Wrapper around HistoryStack
├── flows/
│   └── RecalculationFlow.js     # Orchestrates the edit-recalc flow
└── index.js                     # Main integration point
```

Infrastructure Starter Code

Complete `HistoryStack.js` implementation:

JAVASCRIPT

```javascript
// src/interactions/history/HistoryStack.js

export class HistoryStack {

  constructor(maxDepth = 100) {

    this.maxDepth = maxDepth;

    this.undoStack = [];

    this.redoStack = [];

  }


  push(command) {

    // Clear redo stack when new action is recorded

    this.redoStack = [];



    // Add to undo stack

    this.undoStack.push(command);



    // Enforce max depth by removing oldest commands

    if (this.undoStack.length > this.maxDepth) {

      this.undoStack.shift(); // Remove oldest

    }



    console.log(`Command pushed: ${command.description}. Undo stack: ${this.undoStack.length}, Redo stack:
${this.redoStack.length}`);

  }


  undo() {

    if (this.undoStack.length === 0) {

      console.log("Nothing to undo");

      return null;

    }



    const command = this.undoStack.pop();

    console.log(`Undoing: ${command.description}`);



    try {
```

```
      command.undo();
      this.redoStack.push(command);
      return command;
    } catch (error) {
      // If undo fails, restore command to undo stack
      console.error("Undo failed:", error);
      this.undoStack.push(command);
      throw error;
    }
  }

  redo() {
    if (this.redoStack.length === 0) {
      console.log("Nothing to redo");
      return null;
    }

    const command = this.redoStack.pop();
    console.log(`Redoing: ${command.description}`);

    try {
      command.execute();
      this.undoStack.push(command);
      return command;
    } catch (error) {
      // If redo fails, restore command to redo stack
      console.error("Redo failed:", error);
      this.redoStack.push(command);
      throw error;
    }
  }

  clear() {
```

```
    this.undoStack = [];
    this.redoStack = [];
  }

  clearRedo() {
    this.redoStack = [];
  }

  getUndoCount() {
    return this.undoStack.length;
  }

  getRedoCount() {
    return this.redoStack.length;
  }
}
```

Core Logic Skeletons

Cell edit flow orchestrator:

JAVASCRIPT

```javascript
// src/flows/RecalculationFlow.js

export class RecalculationFlow {

  constructor(spreadsheet, dependencyGraph, formulaParser) {

    this.spreadsheet = spreadsheet;

    this.dependencyGraph = dependencyGraph;

    this.formulaParser = formulaParser;

  }


  /**

   * Main entry point for cell edit with formula

   * @param {number} row - 0-based row index

   * @param {number} col - 0-based column index

   * @param {string} formulaString - Raw formula input (with or without =)

   * @returns {boolean} Success status

   */

  handleCellEdit(row, col, formulaString) {

    // TODO 1: Get cell address from row/col using toCellAddress()



    // TODO 2: Get existing cell to capture previous state for undo



    // TODO 3: If formulaString starts with '=', parse it

    //   - Call parseFormula() with context cell address

    //   - If hasErrors, set cell value to error and return



    // TODO 4: Extract dependencies from parsed formula



    // TODO 5: Remove old dependencies for this cell from graph

    //   - Get current cell's dependencies

    //   - For each, call removeDependency(cellAddress, oldDep)



    // TODO 6: Add new dependencies to graph

    //   - For each new dependency, check for circular reference

    //   - If circular, set error value and return
```

```
    //   - Otherwise addDependency(cellAddress, newDep)


    // TODO 7: Update cell in spreadsheet with new formula
    //   - Call setCellFormula()


    // TODO 8: Get topological order for recalculation
    //   - Start with the edited cell and all its dependents
    //   - Call getTopologicalOrder() for the edited cell


    // TODO 9: Evaluate cells in topological order
    //   - For each cell address in order
    //   - Get its formula, parse if needed
    //   - Evaluate AST with current values
    //   - Update cell value


    // TODO 10: Return success


    return false; // Placeholder
  }
}
```

Reference adjustment implementation:

```javascript
// src/interactions/clipboard/ReferenceAdjuster.js

export function adjustFormula(formula, sourceAddress, targetAddress) {

  // TODO 1: Parse source and target addresses to row/col indices

  //   - Use parseCellAddress() for both



  // TODO 2: Calculate displacement vector

  //   - Δrow = targetRow - sourceRow

  //   - Δcol = targetCol - sourceCol



  // TODO 3: Tokenize the formula to identify cell references

  //   - Use tokenize() from FormulaParser



  // TODO 4: Process each token

  //   - For CELL_REF tokens, parse the referenced address

  //   - Check absolute flags ($A$1, A$1, $A1)

  //   - Adjust relative parts by displacement

  //   - Reconstruct token with adjusted address



  // TODO 5: Handle range tokens (COLON operator)

  //   - Ranges like A1:B2 have start and end references

  //   - Adjust both separately, then reconstruct



  // TODO 6: Reassemble formula string from adjusted tokens



  // TODO 7: Return adjusted formula



  return formula; // Placeholder

}
```

Language-Specific Hints (JavaScript)

- Use `structuredClone()` for deep copying cell state in commands to avoid reference issues.
- Implement `Clipboard API` with fallback: try `navigator.clipboard.writeText()` but catch `SecurityError` and use `document.execCommand('copy')` as fallback.

- For event delegation on the grid, attach a single listener to the container: `gridContainer.addEventListener('click', (e) => { if (e.target.dataset.cellId) { /* handle */ } })`.
- Use `WeakMap` for private fields in command classes if targeting modern browsers.

Milestone Checkpoint: Undo/Redo Implementation

After implementing the undo/redo system:

1. **Run a manual test sequence**:

```
1. Set A1 = 10
2. Set B1 = =A1*2
3. Undo (B1 should become empty)
4. Redo (B1 should show 20)
5. Set A1 = 20 (B1 should auto-update to 40)
6. Undo twice (should get back to initial state)
```

2. **Expected console output** during the test:

```
Command pushed: Set value in A1. Undo stack: 1, Redo stack: 0
Command pushed: Set formula in B1. Undo stack: 2, Redo stack: 0
Undoing: Set formula in B1
Redoing: Set formula in B1
Command pushed: Set value in A1. Undo stack: 3, Redo stack: 0
Undoing: Set value in A1
Undoing: Set formula in B1
```

3. **Signs something is wrong**:
   - Undo/redo doesn't update dependent cells → Missing recalculation in command execution
   - Memory usage climbs indefinitely → Missing `maxDepth` enforcement
   - Can't undo after new action → Redo stack not being cleared

Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
Paste creates wrong references	Displacement calculated incorrectly	Log source/target addresses and displacement vector	Ensure 0-based vs 1-based indexing consistency
Undo causes circular reference error	Dependencies not restored correctly during undo	Check command's `previousState` includes formula, not just value	Store complete cell state in commands
Copy/paste works but dependent cells don't update	Graph dependencies not transferred during paste	Visualize dependency graph before/after paste	Ensure `addDependency()` called for each pasted formula
Redo stack disappears after new edit	`clearRedo()` called at wrong time	Trace when `clearRedo()` is invoked	Only clear redo on new user action, not programmatic changes

Error Handling and Edge Cases

> **Milestone(s):** Milestone 2 (Formula Parser), Milestone 3 (Dependency Graph & Recalculation), Milestone 4 (Advanced Features)

In a spreadsheet system, errors aren't exceptional conditions to be avoided but expected states that must be gracefully managed. Unlike typical applications where errors represent system failures, spreadsheet errors represent semantic issues in user data that require clear communication and recovery pathways. This section documents the taxonomy of expected failure modes, their detection mechanisms, and structured strategies for recovery and user feedback. Proper error handling transforms confusing failures into learning opportunities for users building complex calculations.

Error Type Catalog and Detection

Think of spreadsheet errors as different types of traffic signals in a city's transportation network: some indicate temporary conditions (yield), some warn of danger (caution), and some signal complete stoppage (red light). Each requires different responses from the driver (user) and city maintenance (system). The spreadsheet's error detection system serves as this traffic signaling infrastructure, constantly monitoring for specific patterns that indicate problematic states.

Error Detection Philosophy: Instead of treating errors as exceptions that crash execution, we treat them as first-class values that propagate through calculations. When cell A1 contains `#REF!` and cell B1 contains `=A1+5`, cell B1 automatically becomes `#REF!` too—errors cascade through dependent formulas just like values do, ensuring consistency in the face of partial failures.

The following table catalogs the core error types our spreadsheet must detect and handle:

Error Type	Constant Reference	Detection Trigger	Example Scenario	Detection Method
Circular Reference	`ERRORS.CIRCULAR_REF`	Adding a dependency that creates a cycle	Cell A1 contains `=B1+1` and cell B1 contains `=A1*2`	Graph traversal in `detectCircularReference()` before adding dependency edges
Reference Error	`ERRORS.REF_ERROR`	Referencing non-existent or deleted cell	Formula `=C99` when `MAX_ROWS` is 1000 (valid) but cell doesn't exist yet	`getCellByAddress()` returns `null` during AST evaluation
Value Error	`ERRORS.VALUE_ERROR`	Type mismatch or invalid argument	`=SUM("text", 5)` where text cannot be coerced to number	Type checking in `evaluateNode()` for built-in functions
Division by Zero	`ERRORS.DIV_ZERO`	Dividing any number by zero	`=5/0` or `=A1/B1` where B1 contains 0	Special check in binary operator evaluation for `'/'` operator
Parse Error	`ERRORS.PARSE_ERROR`	Malformed formula syntax	`=5+*3` or `=SUM(A1,)`	Failure in `parseExpression()` due to unexpected token sequence
Range Error	`ERRORS.REF_ERROR`	Invalid cell range specification	`=SUM(A1:Z99)` when `MAX_COLUMNS` is 26 (Z is valid)	Range validation in `resolveRange()` checking bounds
Number Format Error	*None (format-specific)*	Invalid number format string	Setting format to `"currency"` when only `"number"`, `"percent"`, `"currency"` supported	Format validation in `applyFormat()`

Architecture Decision: Error as Value vs. Exception

Decision: Treat Errors as Propagating Values

- **Context**: Formulas can reference other cells that may contain errors, requiring consistent behavior when errors appear in intermediate calculations.
- **Options Considered**:
  1. **Exception throwing**: Immediately throw an exception when any error is encountered, aborting evaluation
  2. **Error objects**: Create error objects that propagate through calculations as special values
  3. **Null propagation**: Return `null` for any calculation involving an error
- **Decision**: Use error objects that propagate through calculations (Option 2)
- **Rationale**: This matches Excel's behavior where `=#REF!+5` results in `#REF!`, not a crashed application. It allows partial evaluation to continue (e.g., parsing succeeds even if evaluation fails) and provides specific error messages.
- **Consequences**: Error checking must occur at every evaluation step, but the system remains stable even with multiple erroneous cells.

The detection mechanisms integrate throughout the system architecture:

Component	Error Detection Points	Detection Method
Formula Parser	Tokenization, parsing, AST construction	Syntax validation, bracket matching, operator placement
Dependency Graph	Edge addition, cycle detection	Graph traversal algorithms (DFS) in `detectCircularReference()`
AST Evaluator	Reference resolution, type checking, arithmetic	Value validation, type coercion checks, boundary conditions
Grid Component	User input validation	Real-time syntax highlighting, immediate feedback on typing

Circular Reference Detection Deep Dive: Circular references represent a particularly challenging error because they threaten system stability (infinite recalculation loops). Our detection uses a modified depth-first search:

1. **Pre-addition check**: Before `addDependency(A, B)` adds edge A→B, we run `detectCircularReference(B, A)` to see if a path already exists from B back to A.
2. **DFS traversal**: Starting from `startAddress` (B), we traverse outgoing edges recursively.
3. **Cycle identification**: If we encounter `targetAddress` (A) during traversal, a cycle would form.
4. **Early termination**: We cache visited nodes and use iterative DFS to handle large graphs efficiently.

The algorithm maintains a `visited` set to avoid redundant traversal and returns `true` as soon as any path from start to target is found.

Recovery and User Feedback Strategies

The key insight for error recovery in spreadsheets is that **errors should be informative, not destructive**. When a user enters `=1/0`, we shouldn't prevent them from typing it—we should let them complete the formula, then show `#DIV/0!` with a helpful explanation. Recovery means providing clear paths to fix the error without losing other work.

User Feedback Design Principles:

1. **Immediacy**: Show errors as soon as they're detectable (during typing or on cell exit)
2. **Specificity**: Provide exact error type and location (e.g., "Circular reference detected: A1 → B1 → C1 → A1")
3. **Preservation**: Never lose user data due to errors—store erroneous formulas for correction

4. **Guidance**: Suggest fixes when possible (e.g., "Did you mean A1 instead of A?")

The recovery strategy varies by error type and context:

Error Type	Immediate UI Feedback	User Recovery Path	System Recovery Action
Circular Reference	Display `#CIRCULAR!` in affected cells with red border	Trace visualization showing the cycle path	Suspend recalculation for cycle, mark cells as "error state", allow editing any cell in cycle to break it
Reference Error	Display `#REF!` in formula cell, highlight invalid reference in formula editor	Edit formula to correct reference or create referenced cell	Treat missing cells as having value `0` during subsequent calculations if user chooses
Parse Error	Red underline under problematic syntax, tooltip with explanation	Edit formula with syntax guidance	Store raw formula string anyway, allow re-editing without loss
Division by Zero	Display `#DIV/0!` with subdued styling (not "error red")	Add IFERROR wrapper: `=IFERROR(A1/B1, 0)`	Continue evaluating other unrelated formulas normally
Value Error	Display `#VALUE!` with indication of which argument failed	Hover tooltip showing expected vs. actual types	Coerce types where safe (string "5" → number 5), error only when impossible

Error State Propagation: When cell A1 contains an error, all cells that depend on A1 should immediately reflect that error state. This propagation occurs during the recalculation phase:

1. Cell A1 evaluates to error object `{type: "#REF!", message: "Invalid cell reference: Z999"}`
2. During topological sort, dependent cell B1 (with formula `=A1+5`) is scheduled for recalculation
3. `evaluateAST()` for B1's formula encounters the error value from A1
4. Instead of performing arithmetic, it returns a new error object with the same type
5. The grid displays `#REF!` in cell B1

Architecture Decision: Error Recovery vs. Prevention

> **Decision: Prioritize Recovery over Prevention**
>
> - **Context**: Users will inevitably create errors while building complex spreadsheets; we must choose between preventing errors at entry time versus allowing them and helping fix them.
> - **Options Considered**:
>   1. **Strict prevention**: Disallow formula submission until syntactically and semantically valid
>   2. **Lenient with warnings**: Allow submission but show warnings
>   3. **Allow with clear errors**: Allow any formula, mark errors clearly, provide repair tools
> - **Decision**: Allow with clear errors (Option 3)
> - **Rationale**: Spreadsheet development is iterative—users often build formulas referencing cells that don't exist yet, or create temporary circular references while rearranging logic. Preventing entry breaks this workflow. Professional spreadsheets (Excel, Sheets) use this approach.
> - **Consequences**: We need robust error display UI and must ensure error cells don't crash the system.

Error Visualization Interface: The `VirtualGrid` component implements these feedback mechanisms:

Visual Element	Purpose	Implementation
Error badge	Small indicator in cell corner showing error type	CSS pseudo-element with error symbol, color-coded by error type
Formula editor highlighting	Underline or highlight the problematic portion of formula	Token-level error tracking in `ParsedFormula.errorPosition`
Error tooltip	Detailed explanation on hover	HTML title attribute or custom tooltip component
Trace visualization	For circular references, show the dependency cycle	Overlay arrows between cells or sidebar panel showing graph
Error sidebar	List all errors in spreadsheet with navigation	Dedicated UI panel populated from error scan

Recovery Actions Table: For each error type, we define specific recovery actions available to users:

Error Context	Recovery Action	Implementation Mechanism
Any error in cell	Edit cell directly	`enterEditMode()` with formula pre-loaded and cursor positioned
Reference to missing cell	Create the missing cell	`setCellValue()` with default value 0, then re-evaluate
Circular reference	Break cycle by editing any cell in cycle	Remove or change the formula in one cell to eliminate dependency
Type mismatch	Wrap with conversion function	Suggest `=VALUE(A1)` for text-to-number conversion
Out of bounds reference	Adjust to valid range	Auto-correct `Z100` to `Z1000` if within `MAX_ROWS`

Edge Cases: Empty Cells, Ranges, and Formula Variants

Beyond explicit errors, spreadsheets must handle numerous edge cases—situations that aren't technically errors but require special handling to match user expectations. Think of these as the "unwritten rules" of spreadsheet behavior that users assume based on experience with Excel or Google Sheets.

Empty Cells Semantics: An empty cell isn't the same as a cell containing `0` or `""`. The system must distinguish between:

1. **Truly empty cells**: Never been edited, don't exist in the `grid` Map
2. **Cleared cells**: Previously had content, now contain empty string `""`
3. **Formula returning empty**: Formula like `=IF(FALSE, 5, "")` evaluates to empty string

The handling differs by context:

Context	Empty Cell Behavior	Implementation
Arithmetic operations	Treated as `0`	`evaluateNode()` returns 0 for `CellReferenceNode` pointing to empty cell
Text concatenation	Treated as empty string `""`	String operations handle empty as zero-length string
SUM() function	Ignored (not counted as 0)	Function implementation filters out empty values
COUNT() function	Not counted	Only cells with numeric values are counted
Logical tests	Treated as `FALSE`	In boolean contexts, empty evaluates to false

Range Edge Cases: Cell ranges like `A1:B5` introduce several subtle behaviors:

Range Scenario	Expected Behavior	Handling Logic
Single-cell range	`A1:A1` should work as valid range	`resolveRange()` returns array with single address
Reverse range	`B5:A1` should normalize to `A1:B5`	Sort start/end by row and column during resolution
Partial emptiness	`SUM(A1:A10)` where A3, A7 are empty	Ignore empty cells, sum only numeric values
Full column/row	Not supported in basic implementation	Return error or limit to `MAX_ROWS` / `MAX_COLUMNS`
Non-rectangular range	Invalid—should be parse error	Validate during `resolveRange()` that start/end form rectangle

Formula Variants and Synonyms: Users expect certain formula variations to work:

Variant Type	Examples	Handling Approach
Case-insensitive functions	`sum()`, `Sum()`, `SUM()` all equivalent	Normalize function names to uppercase during parsing
Spaces in formulas	`= SUM ( A1 )` with arbitrary spaces	Tokenizer ignores whitespace, parser allows flexible spacing
Comma vs. semicolon	Cultural differences in argument separators	Support only comma initially, consistent with US Excel
Implied multiplication	`=5(3+2)` meaning `=5*(3+2)`	Not supported—requires explicit `*` operator
Leading plus	`=+A1+5` (historical Excel syntax)	Parser treats leading `+` as no-op, allowed but ignored

Architecture Decision: Strict vs. Lenient Parsing

Decision: Balanced Leniency with Clear Rules

- **Context**: Users have diverse backgrounds and may import formulas from other spreadsheet systems with slightly different syntax.
- **Options Considered**:
  1. **Strict parsing**: Accept only exactly correct syntax matching our grammar
  2. **Very lenient**: Accept many variants, auto-correcting where possible
  3. **Balanced**: Accept common real-world variants but reject truly malformed formulas
- **Decision**: Balanced approach (Option 3)
- **Rationale**: We want to be compatible with user expectations from major spreadsheets but not create an unpredictable "magic" parser that guesses wrong. Case-insensitivity and spacing flexibility have no downside, while auto-correction of `5(3+2)` to `5*(3+2)` could hide user errors.
- **Consequences**: Parser must explicitly handle each allowed variant, increasing complexity slightly, but user experience improves.

Number Formatting Edge Cases: When cells have both values and formatting, edge cases emerge:

Scenario	Behavior	Implementation
Format applied to empty cell	Format stored, applies when cell gets value	`Cell.format` persists independently of `Cell.value`
Percentage of already formatted number	`=A1*10%` where A1 is formatted as currency	Format doesn't affect stored value, only display
Rounding vs. display precision	Cell shows `3.33` but contains `3.333333`	`decimalPlaces` in `CellFormat` affects display only
Format removal	Clear format should show raw stored value	Set `Cell.format` to `null` or default

Undo/Redo with Errors: Error states must be properly captured in the undo/redo history:

1. **Error introduction**: When a formula change creates an error, the `SetCellFormulaCommand` stores both the previous (possibly valid) state and the new (error) state.
2. **Error propagation**: When undoing, we must revert not just the changed cell but all dependent cells that showed propagated errors.
3. **Compound errors**: Multiple cells becoming erroneous from one change should be grouped in a `CompoundCommand` for single undo step.
4. **Error state visualization in history**: The undo stack should indicate which commands introduced errors.

Cross-Milestone Edge Case Integration:

Milestone	Edge Cases Affected	Coordination Required
Milestone 1 (Grid)	Displaying error indicators in cells	`VirtualGrid` must read `Cell.value` error types for styling
Milestone 2 (Parser)	Syntax error recovery, malformed formulas	`parseFormula()` must return `ParsedFormula.hasErrors` with position
Milestone 3 (Graph)	Circular reference detection, error propagation	`DependencyGraph` must track error states through dependencies
Milestone 4 (Advanced)	Copy/paste of error cells, CSV export of errors	`ClipboardManager` must preserve error types, CSV must encode errors

The system's approach to these edge cases follows the principle of **"predictable generosity"**—generous in accepting user input in various forms, but perfectly predictable in how that input is interpreted and displayed. This balance makes the spreadsheet both approachable for beginners and reliable for complex work.

Implementation Guidance

Technology Recommendations:

Component	Simple Option	Advanced Option
Error Display	CSS classes + `title` attributes for tooltips	Custom SVG error indicators with detailed hovercards
Error Propagation	Simple type checking in evaluator	Error type hierarchy with custom propagation rules
Circular Reference Detection	Recursive DFS with visited set	Iterative BFS with cycle path reconstruction

Recommended File Structure:

```
src/
├── errors/
│   ├── ErrorTypes.js          # Error constants and type definitions
│   ├── ErrorPropagator.js     # Logic for error propagation through formulas
│   └── ErrorDisplay.js        # UI components for error visualization
├── utils/
│   └── validation.js          # Validation utilities for ranges, addresses
└── components/
    └── ErrorSidebar.js        # Sidebar listing all spreadsheet errors
```

Infrastructure Starter Code (complete error handling utilities):

JAVASCRIPT

```javascript
// src/errors/ErrorTypes.js

export const ERRORS = {

  CIRCULAR_REF: '#CIRCULAR!',

  REF_ERROR: '#REF!',

  VALUE_ERROR: '#VALUE!',

  DIV_ZERO: '#DIV/0!',

  PARSE_ERROR: '#PARSE!',



  isError(value) {

    return typeof value === 'string' && value.startsWith('#') && value.endsWith('!');

  },



  getMessage(errorCode) {

    const messages = {

      '#CIRCULAR!': 'Circular reference detected',

      '#REF!': 'Invalid cell reference',

      '#VALUE!': 'Invalid value or type',

      '#DIV/0!': 'Division by zero',

      '#PARSE!': 'Formula parse error'

    };

    return messages[errorCode] || 'Unknown error';

  }

};



// src/errors/ErrorPropagator.js

export class ErrorPropagator {

  // Propagate error through binary operations

  static propagateBinary(left, right, operator) {

    if (ERRORS.isError(left)) return left;

    if (ERRORS.isError(right)) return right;



    // Actual operation logic would go here

    // For errors, we've already returned above
```

```
    return null; // Not an error case
  }

  // Propagate error through function calls
  static propagateFunction(args) {
    for (const arg of args) {
      if (ERRORS.isError(arg)) return arg;
    }
    return null; // No errors in arguments
  }
}

// src/utils/validation.js
export function isValidRange(startRow, startCol, endRow, endCol) {
  return startRow >= 0 && startRow <= endRow && endRow < MAX_ROWS &&
         startCol >= 0 && startCol <= endCol && endCol < MAX_COLUMNS;
}

export function normalizeAddress(address) {
  // Convert various address formats to standard A1 format
  const match = address.toUpperCase().match(/(\$?[A-Z]+)(\$?[0-9]+)/);
  return match ? `${match[1]}${match[2]}` : null;
}
```

Core Logic Skeleton Code:

JAVASCRIPT

```javascript
// src/dependency/DependencyGraph.js - Circular reference detection

detectCircularReference(startAddress, targetAddress) {

  // TODO 1: If startAddress equals targetAddress, return true immediately (self-reference)

  // TODO 2: Initialize visited Set with startAddress to prevent infinite loops

  // TODO 3: Initialize stack/queue for DFS/BFS traversal starting from startAddress

  // TODO 4: While there are nodes to process:

  // TODO 5:   Get current node from stack/queue

  // TODO 6:   Retrieve all outgoing dependencies (edges) from current node

  // TODO 7:   For each dependent cell address:

  // TODO 8:     If dependent equals targetAddress, return true (cycle found!)

  // TODO 9:     If dependent not visited yet, add to visited and push to stack/queue

  // TODO 10: If traversal completes without finding target, return false (no cycle)

  // Hint: Use iterative DFS with explicit stack for large graphs

}


// src/parser/FormulaEvaluator.js - Error-aware evaluation

evaluateNode(node, valueResolver) {

  // TODO 1: Handle ErrorNode type - return error value directly

  // TODO 2: Handle CellReferenceNode - resolve cell value via valueResolver

  // TODO 3: If resolved value is an error (ERRORS.isError()), return that error

  // TODO 4: Handle NumberNode - return numeric value

  // TODO 5: Handle BinaryOpNode - evaluate left and right children first

  // TODO 6: Check if either child evaluation returned error, propagate if so

  // TODO 7: For division operator, check if right child evaluates to 0

  // TODO 8: If division by zero, return ERRORS.DIV_ZERO

  // TODO 9: Otherwise perform operation and return result

  // TODO 10: Handle FunctionCallNode - evaluate all arguments first

  // TODO 11: Check any argument errors, propagate if found

  // TODO 12: Validate argument types/counts for built-in functions

  // TODO 13: If validation fails, return ERRORS.VALUE_ERROR

  // TODO 14: Execute function with validated arguments

}
```

Language-Specific Hints:

- Use `Number.isNaN()` instead of global `isNaN()` for proper NaN detection
- When checking for empty cells, distinguish between `undefined`, `null`, `""`, and `0`
- Use `Map` and `Set` for graph operations for O(1) lookups
- Implement `valueOf()` or `toString()` on error objects for natural propagation

Milestone Checkpoint: After implementing error handling:

1. Run test: `node test-errors.js` (hypothetical test file)
2. Expected output: All error tests pass, showing proper detection and propagation
3. Manual verification:
   - Enter `=1/0` in any cell → Should display `#DIV/0!` without crashing
   - Create circular reference A1: `=B1` and B1: `=A1` → Should show `#CIRCULAR!` in both
   - Reference non-existent cell `=Z1001` → Should show `#REF!`
   - Fix the error → Cells should update immediately

Debugging Tips:

Symptom	Likely Cause	How to Diagnose	Fix
Errors not showing in UI	Grid not reading error flags	Check `VirtualGrid.renderCell()` for error class addition	Add error CSS class when `ERRORS.isError(cell.value)`
Circular ref detection too slow	Exponential recursion on large graphs	Add visited set logging	Use iterative DFS with explicit stack
Error propagates incorrectly	Missing error check in operator	Add console.log to `ErrorPropagator`	Ensure all operations check for error inputs first
Undo doesn't revert errors	Error state not captured in command	Inspect `SetCellFormulaCommand` serialization	Include error state in `previousState` / `newState`

Testing Strategy

> **Milestone(s):** Milestone 1, 2, 3, 4 (Testing is cross-cutting and applies to all milestones)

A comprehensive testing strategy is essential for building a reliable spreadsheet engine. Unlike simpler applications, spreadsheets involve complex interactions between parsing, dependency tracking, recalculation, and UI rendering—where bugs in one component can cascade through the entire system. This section outlines a multi-layered testing approach that combines unit tests for isolated component behavior, integration tests for cross-component workflows, and milestone verification checkpoints to validate end-to-end functionality. The goal is to catch errors early, particularly in the dependency graph and formula evaluation where logical errors can be subtle but catastrophic.

Testing Approach by Component

Think of testing a spreadsheet as similar to testing a complex mechanical watch. Each gear (component) must work perfectly in isolation, but the real test is whether they mesh correctly to tell accurate time. We'll test each gear (parser, graph, grid) independently, then assemble them and verify the entire mechanism keeps correct time.

Virtualized Grid Component Testing

The virtualized grid presents unique testing challenges because it combines DOM manipulation, viewport calculations, and user interaction. Testing focuses on three areas: rendering correctness, scrolling behavior, and selection/editing workflows.

Rendering Correctness Tests:

Test Scenario	Input/Setup	Expected Behavior	Verification Method
Initial render with empty spreadsheet	Mount `VirtualGrid` with empty `Spreadsheet`	Grid displays exactly `VIEWPORT_ROWS` × `VIEWPORT_COLUMNS` cells with correct row/column headers	DOM inspection for cell count and header text
Cell value rendering	Set cell A1 to "Test" via `setCellValue(0,0,"Test")`	Cell at position (0,0) displays "Test" when in viewport	DOM text content check
Formula cell display	Set cell B2 to formula "=A1+5" with A1=10	Cell B2 displays "15" (evaluated result) not "=A1+5"	DOM text content equals evaluated value
Format application	Apply `bold:true` format to cell C3	Cell C3's font-weight is "bold" when rendered	Computed style inspection
Empty viewport areas	Scroll far beyond populated cells	Empty cells show no content but maintain grid structure	DOM elements exist with empty text

Virtual Scrolling Algorithm Tests: The scrolling logic is critical for performance. We test viewport coordinate translation and DOM recycling:

1. **Viewport Translation Test:** Simulate scroll events and verify `renderViewport()` updates the correct logical cells.
2. **DOM Recycling Test:** Verify that scrolling doesn't create new DOM elements indefinitely by tracking element count.
3. **Scroll Boundary Test:** Ensure scrolling beyond `MAX_ROWS` or `MAX_COLUMNS` shows appropriate empty space.

Selection and Editing Tests:

Test Scenario	User Action	Expected Grid Response
Click selection	Click cell at (5,2)	`setSelection(5,2)` called, cell gets selection CSS class
Keyboard navigation	Press ArrowRight from selected cell	Selection moves to adjacent cell, viewport scrolls if needed
Double-click edit	Double-click cell with value "Hello"	`enterEditMode()` shows input with "Hello", cursor positioned
Edit completion	Type "New" and press Enter	Input disappears, cell displays "New", `handleCellEdit` triggered

> **Testing Insight:** Mock the `Spreadsheet` data model completely for grid tests—the grid should only depend on the data model interface, not actual implementation. This allows testing the grid's rendering logic independently of formula evaluation or dependency tracking.

Formula Parser and Evaluator Testing

The formula parser transforms raw strings into executable expression trees. Testing requires both syntax validation (correct formulas parse correctly) and semantic validation (formulas evaluate to correct values).

Tokenization Tests: The `tokenize()` function must correctly identify all token types:

Input String	Expected Tokens (Type, Value)
`"=A1+5"`	`CELL_REF "A1"`, `OPERATOR "+"`, `NUMBER "5"`
`"=SUM(A1:A10)"`	`FUNCTION "SUM"`, `LEFT_PAREN "("`, `CELL_REF "A1"`, `COLON ":"`, `CELL_REF "A10"`, `RIGHT_PAREN ")"`
`"= -3.14 * B2"`	`OPERATOR "-"`, `NUMBER "3.14"`, `OPERATOR "*"`, `CELL_REF "B2"`
Edge: `"=A1"` (trailing space)	`CELL_REF "A1"`, `EOF` (ignore trailing whitespace)

AST Construction Tests: Verify `buildAST()` creates correct tree structures respecting operator precedence:

Formula	Expected AST Structure (Simplified)
`"=1+2*3"`	BinaryOpNode('+', NumberNode(1), BinaryOpNode('*', NumberNode(2), NumberNode(3)))
`"=(1+2)*3"`	BinaryOpNode('*', BinaryOpNode('+', NumberNode(1), NumberNode(2)), NumberNode(3))
`"=SUM(A1,B2)"`	FunctionCallNode('SUM', [CellReferenceNode('A1'), CellReferenceNode('B2')])
`"=-A1"`	UnaryOpNode('-', CellReferenceNode('A1'))

Evaluation Tests: The `evaluateAST()` function must correctly compute values given a value resolver:

Test Setup	Formula	Expected Result
A1=10, B1=20	`"=A1+B1"`	30
A1=5	`"=A1*2+3"`	13
A1=0	`"=1/A1"`	`ERRORS.DIV_ZERO`
Empty cell referenced	`"=A1+5"`	5 (treat empty as 0)
Range A1:A3 = [1,2,3]	`"=SUM(A1:A3)"`	6
Circular reference detected	`"=B1"` where B1 references this cell	`ERRORS.CIRCULAR_REF`

Edge Case Tests:

- **Mixed references:** `"=$A$1+B$2+$C3"` parses with correct absolute flags
- **Function nesting:** `"=SUM(A1, MAX(B1:B5))"` builds correct nested AST
- **Whitespace variations:** Formulas with extra spaces should still parse
- **Error propagation:** Parse errors in subexpressions bubble up appropriately

Architecture Decision Record: Testing Strategy for Parser

- **Context:** The parser must handle complex Excel-like syntax while providing clear error messages. Testing needs to cover both valid formulas and numerous invalid cases.
- **Options Considered:**
  1. **End-to-end integration tests only:** Test entire formula evaluation through UI.
  2. **Comprehensive unit tests for each parsing stage:** Test tokenizer, parser, evaluator independently.
  3. **Property-based testing:** Generate random formulas and compare with known evaluators.
- **Decision:** Comprehensive unit tests for each parsing stage (Option 2).
- **Rationale:** Integration tests are too coarse-grained to pinpoint where parsing fails. Property-based testing is valuable but requires a reference implementation. Unit tests allow precise failure isolation and documentation of expected behavior for each syntax construct.
- **Consequences:** Large test suite for parser (~50-100 test cases), but failures immediately point to specific tokenization, parsing, or evaluation bugs. Test maintenance burden when extending syntax.

Dependency Graph and Recalculation Engine Testing

The dependency graph manages the most critical spreadsheet invariant: dependencies must be acyclic, and recalculation must follow dependency order. Tests must validate graph mutations and topological sort correctness.

Graph Mutation Tests:

Test Scenario	Method Call	Expected Graph State	Invariants to Check
Add single dependency	`addDependency("B1", "A1")`	Edge A1→B1 added, B1's `dependencies` includes "A1"	No cycles created
Remove dependency	`removeDependency("B1", "A1")`	Edge A1→B1 removed, B1's `dependencies` excludes "A1"	Graph remains consistent
Multiple dependencies	B1 depends on A1 and C1	B1 has both in dependency set	`getDependents("A1")` includes B1
Transitive dependency chain	A1→B1→C1	Graph has edges A1→B1, B1→C1	Topological order: [A1, B1, C1]

Circular Reference Detection Tests: The `detectCircularReference()` function must identify cycles before they corrupt the graph:

Existing Graph	Proposed Edge	Expected Detection Result
Empty	A1→A1	True (self-reference)
A1→B1	B1→A1	True (2-cycle)
A1→B1, B1→C1	C1→A1	True (3-cycle)
A1→B1	B1→C1	False (no cycle)

Topological Sort Tests: `getTopologicalOrder()` must produce valid evaluation orders:

Graph Structure	Valid Topological Orders
A1→B1, A1→C1	[A1, B1, C1] or [A1, C1, B1]
A1→B1→C1, A1→D1	[A1, B1, D1, C1] or [A1, D1, B1, C1] (B1 before C1)
Independent cells A1, B1, C1	Any permutation of [A1, B1, C1]
Complex diamond: A1→B1, A1→C1, B1→D1, C1→D1	A1 before B1/C1, B1/C1 before D1

Recalculation Integration Tests: These tests combine the graph with the parser and evaluator:

1. **Basic Recalculation Flow:** Set A1=5, B1=`=A1*2`, change A1 to 10, verify B1 becomes 20.
2. **Transitive Recalculation:** A1=1, B1=`=A1+1`, C1=`=B1*2`, change A1→C1 updates in correct order.
3. **Partial Recalculation:** When A1 changes, only cells depending on A1 recalculate (verify other cells not touched).
4. **Error Propagation:** If A1 has `#DIV/0!`, dependent cells show appropriate error.

> **Testing Insight:** Use directed graph visualization for debugging test failures. When a topological sort test fails, draw the graph to verify expected dependencies match actual.

Advanced Features Manager Testing

Advanced features involve stateful operations (undo/redo), reference adjustment logic, and file format handling. Testing requires simulating user workflows and verifying state transitions.

Undo/Redo Tests: The `HistoryStack` and command pattern must maintain consistent state:

Operation Sequence	Expected Undo/Redo State
Set A1=5, Set B1=10	Undo once: B1 reverts to previous value
Two undos then new edit	Redo stack cleared (branching timeline)
Compound command (paste range)	Single undo reverts all pasted cells
Maximum stack depth (e.g., 50)	Oldest commands discarded when exceeding limit

Copy/Paste with Reference Adjustment Tests: The `adjustFormula()` function must correctly transform relative and absolute references:

Original Formula (in A1)	Paste Destination	Adjusted Formula
`=B1` (relative)	C3 (2 rows down, 2 cols right)	`=D3`
`=$B$1` (absolute)	C3	`=$B$1` (unchanged)
`=B$1` (mixed)	C3	`=D$1` (column adjusts, row fixed)
`=SUM(A1:A10)`	C3	`=SUM(C3:C12)` (range shifts)
`=SUM($A$1:$A$10)`	C3	`=SUM($A$1:$A$10)` (absolute range)

CSV Import/Export Tests: Round-trip consistency is critical—exporting then re-importing should preserve data:

Spreadsheet Content	Exported CSV	Re-imported Result
Simple values A1="Hello", B1=42	`"Hello",42`	Identical values
Formulas	Option: export formulas as text or values	Configurable behavior
Empty cells	`,,` (consecutive commas)	Preserve empty cells
Values with commas/quotes	`"Value, with comma"`	Proper escaping

Formatting Tests: Verify `applyFormat()` and `removeFormat()` modify cell metadata correctly and trigger re-rendering.

Milestone Verification Checkpoints

Each milestone has specific acceptance criteria that require both automated tests and manual verification. These checkpoints provide confidence that the system meets functional requirements before proceeding.

Milestone 1: Grid & Cell Rendering Verification

> **Mental Model:** Testing the grid is like testing a theater stage—the lights (selection), backdrop (headers), and seating (cells) must all work before the play (spreadsheet logic) begins.

Acceptance Criteria	Verification Method	Test Cases
Grid renders at least 26 columns and 100 rows with scrollable viewport	**Automated Test:** Check DOM element count matches `VIEWPORT_ROWS` × `VIEWPORT_COLUMNS` and scrollbars appear. **Manual Check:** Scroll horizontally/vertically and observe smooth movement.	- Render empty grid - Scroll to column Z - Scroll to row 100
Double-clicking a cell enters edit mode with cursor in inline text input	**Automated Test:** Simulate double-click event and verify input element appears with focus. **Manual Check:** Double-click various cells, type, confirm input works.	- Double-click empty cell - Double-click cell with existing value - Edit, then press Enter/Escape
Selected cell is visually distinct with highlighted border or background color	**Automated Test:** Check CSS class application after `setSelection()`. **Manual Check:** Click and keyboard navigate, verify highlight moves.	- Click cell A1 (highlight appears) - Press arrow keys (highlight moves) - Click another cell
Scrolling pans the grid while keeping row and column headers visible	**Automated Test:** Verify headers remain in DOM with fixed positioning after scroll events. **Manual Check:** Scroll extensively, headers stay visible.	- Scroll down 50 rows (row headers stay) - Scroll right 10 columns (column headers stay)

Milestone 1 Checkpoint Procedure:

1. Run grid unit tests: `npm test -- VirtualGrid.test.js`
2. Expected: All tests pass (green).
3. Manual verification checklist:
   - Open the spreadsheet in browser
   - Confirm 26 column headers (A-Z) and 100+ row numbers

- Click and drag scrollbars—viewport moves, headers remain fixed
- Double-click any cell, type "Test", press Enter—cell displays "Test"
- Use arrow keys to move selection—highlight follows keypress
- Resize browser window—grid adapts without visual corruption

Milestone 2: Formula Parser Verification

Acceptance Criteria	Verification Method	Test Cases
Formulas starting with = are parsed into evaluable expression trees	**Automated Test:** Feed formulas to `parseFormula()` and validate AST structure. **Manual Check:** Enter formula in UI, see calculated result.	- `=1+2` → AST with BinaryOpNode - `=SUM(A1:A3)` → FunctionCallNode with RangeNode
Cell references like A1 and B2 resolve to their current numeric values	**Automated Test:** Mock value resolver returns values for referenced cells. **Manual Check:** Set A1=5, B1= `=A1*2` , verify B1=10.	- Direct reference `=A1` - Multiple references `=A1+B2` - Chained references `=A1` where A1= `=B1`
Arithmetic operators +, -, *, / evaluate with correct precedence and associativity	**Automated Test:** Test operator precedence with varied expressions. **Manual Check:** Enter complex formulas, verify correct calculation.	- `=1+2*3` → 7 not 9 - `=10-2-3` → 5 (left associative) - `=2^3^2` → 512 (right associative if implemented)
SUM(A1:A10) correctly sums all values in the specified cell range	**Automated Test:** Mock range with values 1..10, verify SUM=55. **Manual Check:** Populate range, enter SUM formula.	- Empty range → 0 - Mixed numbers/text → ignore text - Partial range with errors → error propagation

Milestone 2 Checkpoint Procedure:

1. Run parser unit tests: `npm test -- FormulaParser.test.js`
2. Expected: 100+ tests pass covering tokenization, parsing, evaluation.
3. Manual verification:
   - In UI, type `=1+2*3` in any cell → shows 7
   - Set A1=10, A2=20, in A3 type `=SUM(A1:A2)` → shows 30
   - Test nested functions: `=SUM(1, MAX(2,3))` → 4
   - Enter invalid formula `=1+` → shows parse error indicator
4. Integration test: Create small spreadsheet with multiple formulas, verify all compute correctly.

Milestone 3: Dependency Graph & Recalculation Verification

Acceptance Criteria	Verification Method	Test Cases
Changing a cell value triggers recalculation of all dependent cells automatically	**Automated Test:** Set up dependency chain, mock change, verify dependent cells recalculate. **Manual Check:** Change source cell, watch dependent cells update.	- A1=5, B1=`=A1*2`, C1=`=B1+1` → change A1 to 10, B1→20, C1→21
Topological sort ensures cells are recalculated after their dependencies update	**Automated Test:** Verify recalculation order matches topological sort. **Manual Check:** Log recalculation order for complex graph.	- Diamond dependency: A1→{B1,C1}→D1 → B1/C1 before D1
Circular reference is detected and displays error instead of hanging indefinitely	**Automated Test:** Attempt to create cycle, verify error thrown and graph unchanged. **Manual Check:** Enter circular formula, see error message.	- Direct: A1=`=A1` → #CIRCULAR_REF - Indirect: A1=`=B1`, B1=`=A1` → error
Only cells directly or transitively depending on changed cell are recalculated	**Automated Test:** Instrument recalculation counter, verify unaffected cells skip evaluation. **Manual Check:** Change isolated cell, verify unrelated formulas untouched.	- Change A1 when only B1 depends on it → C1 (independent) not recalculated

Milestone 3 Checkpoint Procedure:

1. Run graph unit tests: `npm test -- DependencyGraph.test.js`
2. Run integration tests: `npm test -- RecalculationFlow.test.js`
3. Expected: All tests pass, particularly cycle detection and topological sort.
4. Manual verification of complex scenarios:
   - Create formula network with 10+ cells
   - Change root cell, observe propagation (consider visual dependency graph debugger)
   - Attempt to create circular reference → error appears immediately
   - Performance: Change one cell in large dependency tree (~100 cells) → updates within 100ms
5. Stress test: Chain of 1000 dependencies, ensure recalculation completes without stack overflow.

Milestone 4: Advanced Features Verification

Acceptance Criteria	Verification Method	Test Cases
Pasting a formula adjusts relative cell references to match new position	**Automated Test:** Test `adjustFormula()` with various reference types. **Manual Check:** Copy formula cell, paste elsewhere, verify adjustment.	- Copy `=A1` from B2 to C3 → becomes `=B2`
Undo reverts the most recent cell change and redo reapplies it correctly	**Automated Test:** Command stack operations with multiple undo/redo. **Manual Check:** Make edits, undo, redo, verify state restoration.	- Edit A1, B1, undo twice, redo once
CSV export produces valid comma-separated file openable by other spreadsheet programs	**Automated Test:** Round-trip test: export → parse → compare. **Manual Check:** Export CSV, open in Excel/LibreOffice.	- Simple grid export - Special characters and commas - Formulas vs values option
Number formatting displays cells as currency, percentage, or fixed decimal places	**Automated Test:** Apply format, verify cell metadata and display text. **Manual Check:** Apply currency format, see $ symbol.	- 1.5 with currency → "$1.50" - 0.05 with percent → "5%"

Milestone 4 Checkpoint Procedure:

1. Run advanced features tests: `npm test -- AdvancedFeaturesManager.test.js`
2. Expected: Tests for undo/redo, copy/paste, CSV, formatting all pass.
3. Manual end-to-end workflow tests:
   - **Copy/Paste:** Copy range with mixed formulas, paste elsewhere, verify adjusted formulas
   - **Undo/Redo:** Make 5 edits, undo 3, make new edit (clears redo), verify consistency
   - **CSV Export:** Create spreadsheet, export CSV, import into new sheet, verify data matches
   - **Formatting:** Apply number formats, verify display, then clear formatting
4. Integration with previous milestones: Ensure advanced features don't break core formula/recalculation functionality.

Implementation Guidance

Technology Recommendations:

Component	Simple Option (for learning)	Advanced Option (for production)
Testing Framework	Jest (JavaScript) or pytest (Python)	Same, with increased coverage metrics
Test Runner	Built-in test runner (Node.js test runner)	Continuous integration (GitHub Actions)
Mocking	Manual mocks and spies	Jest automatic mocking / sinon
UI Testing	Manual verification + DOM unit tests	Playwright or Cypress for E2E
Performance Testing	Manual timing in tests	Benchmark.js with regression detection

Recommended File/Module Structure for Tests:

```
spreadsheet-engine/
├── src/
│   ├── components/
│   │   ├── VirtualGrid/
│   │   │   ├── VirtualGrid.js
│   │   │   └── VirtualGrid.test.js      # Grid unit tests
│   │   ├── FormulaParser/
│   │   │   ├── FormulaParser.js
│   │   │   ├── Tokenizer.js
│   │   │   └── FormulaParser.test.js    # Parser unit tests
│   │   └── DependencyGraph/
│   │       ├── DependencyGraph.js
│   │       └── DependencyGraph.test.js  # Graph unit tests
│   ├── data-model/
│   │   ├── Spreadsheet.js
│   │   └── Spreadsheet.test.js          # Data model tests
│   └── advanced-features/
│       ├── AdvancedFeaturesManager.js
│       └── AdvancedFeaturesManager.test.js
├── integration/
│   └── RecalculationFlow.test.js        # Cross-component integration
├── e2e/
│   └── spreadsheet-ui.test.js           # End-to-end UI tests (optional)
└── package.json
```

Test Infrastructure Starter Code: For JavaScript/Jest, a basic test setup with helpers for spreadsheet testing:

JAVASCRIPT

```javascript
// test-helpers.js

/**
 * Creates a mock spreadsheet with predefined cell values
 * @param {Array} cellDefinitions - Array of {address, value, formula}
 * @returns {Spreadsheet} Mock spreadsheet instance
 */
export function createMockSpreadsheet(cellDefinitions = []) {
  const spreadsheet = createSpreadsheet();
  cellDefinitions.forEach(({ address, value, formula }) => {
    const [col, row] = parseCellAddress(address);
    if (formula) {
      setCellFormula(row, col, formula);
    } else {
      setCellValue(row, col, value);
    }
  });
  return spreadsheet;
}

/**
 * Mock value resolver for formula evaluation tests
 * @param {string} address - Cell address to resolve
 * @returns {any} The cell's current value or 0 if empty
 */
export function mockValueResolver(address) {
  // This would be replaced with actual spreadsheet lookup in integration tests
  return 0;
}

/**
 * Returns all cell addresses in a range (inclusive)
 * @param {string} start - Start address (e.g., 'A1')
 * @param {string} end - End address (e.g., 'C3')
```

```
 * @returns {Array<string>} Array of cell addresses
 */
export function expandRange(start, end) {
  const startRef = parseCellAddress(start);
  const endRef = parseCellAddress(end);
  const addresses = [];
  for (let row = startRef.row; row <= endRef.row; row++) {
    for (let col = startRef.column; col <= endRef.column; col++) {
      addresses.push(toCellAddress(row, col, false, false));
    }
  }
  return addresses;
}
```

Core Test Skeletons: For each component, write tests that follow the algorithm steps described in design:

JAVASCRIPT

```javascript
// VirtualGrid.test.js - Example test structure

describe('VirtualGrid', () => {

  let grid, spreadsheet, container;


  beforeEach(() => {

    container = document.createElement('div');

    spreadsheet = createMockSpreadsheet([

      { address: 'A1', value: 'Test Value' },

      { address: 'B2', formula: '=A1*2' }

    ]);

    grid = new VirtualGrid(container, spreadsheet);

    grid.initializeGrid();

  });


  test('renders correct number of visible cells', () => {

    // TODO 1: Count cell elements in container

    // TODO 2: Assert count equals VIEWPORT_ROWS * VIEWPORT_COLUMNS

    // TODO 3: Verify row and column headers exist

  });


  test('updates cell display when value changes', () => {

    // TODO 1: Change spreadsheet cell value via setCellValue

    // TODO 2: Call grid.renderViewport()

    // TODO 3: Verify DOM element displays new value

  });


  // Additional tests for scrolling, selection, editing...

});


// FormulaParser.test.js - Operator precedence test

describe('FormulaParser operator precedence', () => {

  test('multiplication before addition', () => {

    const parser = new FormulaParser();

    const result = parser.parseFormula('=1+2*3', 'A1');

    // TODO 1: Evaluate the parsed formula with mock resolver
```

```
    // TODO 2: Assert result equals 7 (not 9)
  });
});

// DependencyGraph.test.js - Circular reference detection
describe('DependencyGraph cycle detection', () => {
  let graph;

  beforeEach(() => {
    graph = new DependencyGraph();
    graph.addDependency('B1', 'A1'); // B1 depends on A1
  });

  test('detects direct circular reference', () => {
    // TODO 1: Call detectCircularReference('A1', 'B1') (would create A1←B1)
    // TODO 2: Assert returns true (cycle would be created)
    // TODO 3: Verify graph unchanged after detection
  });
});
```

Milestone Checkpoint Commands: After implementing each milestone, run these verification steps:

1. **Milestone 1 Checkpoint:**

```bash
npm test -- VirtualGrid.test.js
# Expected: 10-15 passing tests
# Then open browser: http://localhost:3000 (or your dev server)
# Manually verify scrolling, editing, selection
```

2. **Milestone 2 Checkpoint:**

```bash
npm test -- FormulaParser.test.js
# Expected: 50+ passing tests covering tokenization, parsing, evaluation
# Manual: Test formulas in UI, verify calculations match expectations
```

3. **Milestone 3 Checkpoint:**

```bash
npm test -- DependencyGraph.test.js RecalculationFlow.test.js

# Expected: 30+ passing tests for graph operations and integration

# Manual: Create dependency chain, change source, verify propagation
```

4. **Milestone 4 Checkpoint:**

```bash
npm test -- AdvancedFeaturesManager.test.js

# Expected: 40+ passing tests for undo/redo, copy/paste, CSV, formatting

# Manual: Complete end-to-end workflows (copy-paste formulas, undo chain, export/import)
```

Debugging Tips for Test Failures:

Symptom	Likely Cause	How to Diagnose	Fix
Formula tests fail with "undefined" errors	Tokenizer not handling edge cases	Add console.log to tokenize() output	Extend token regex patterns
Circular reference detection false positives	Graph adjacency incorrectly updated	Visualize graph with `console.log(graph.edges)`	Verify addDependency/removeDependency logic
Undo/redo loses cell formatting	Shallow copy of cell state	Check Command object's deep copy implementation	Implement proper deep copy for Cell objects
CSV export missing formulas	Export configured for values only	Check `ExportOptions.includeFormulas` flag	Pass correct options to `exportToCSV()`
Recalculation order wrong	Topological sort algorithm error	Log sort output and compare with hand-calculated order	Fix Kahn's algorithm implementation (indegree tracking)

Debugging Guide

> **Milestone(s):** Milestone 1 (Grid & Cell Rendering), Milestone 2 (Formula Parser), Milestone 3 (Dependency Graph & Recalculation), Milestone 4 (Advanced Features) — Debugging is a cross-cutting concern that applies to all milestones

Debugging a spreadsheet application presents unique challenges due to its **reactive dataflow** nature and the complex interactions between components. When a formula in cell A1 changes, the effects ripple through the dependency graph, causing cascading recalculations that can lead to unexpected behavior far from the original change. This section provides a systematic approach to identifying and resolving common bugs, along with specialized techniques for visualizing and tracing spreadsheet-specific behavior.

Think of the spreadsheet as a **complex plumbing system** where formulas are pipes connecting cells. When water (data) flows through these pipes, a leak or blockage in one section can cause issues downstream. The debugging techniques in this section help you install "pressure gauges" and "flow meters" throughout the system to pinpoint exactly where problems occur.

Common Bug Symptoms and Fixes

This table categorizes the most frequently encountered issues during spreadsheet implementation, organized by the milestone where they typically appear. Each entry describes the observable symptom, explains the underlying cause, and provides specific steps to diagnose and fix the problem.

Symptom	Likely Cause	How to Diagnose	Fix
Grid Rendering: Cells disappear or jump during scrolling	The **virtual scrolling** algorithm incorrectly recycles DOM elements, placing them at wrong positions in the viewport. The `renderViewport()` method fails to update cell content or position properly during scroll events.	1. Log the `currentViewport` state (topRow, leftCol) during scroll events 2. Check if `CELL_HEIGHT` and `CELL_WIDTH` constants match actual CSS dimensions 3. Verify DOM recycling logic: each cell element should have its row/col attributes updated before being repositioned	Update the `renderViewport()` method to: (1) Calculate absolute pixel positions based on row/col indices, (2) Ensure cell pool elements are reassigned to new logical positions before CSS transformation, (3) Reset cell content completely when reassigning
Selection highlighting appears on wrong cell	The `setSelection()` method uses incorrect coordinate mapping between logical grid positions and visual viewport positions. The selection state doesn't account for scroll offset.	1. Compare the spreadsheet's `selection` state with the `viewport` offset 2. Check if selection events fire with correct row/col parameters 3. Inspect CSS transforms applied to selected cell element	Modify `setSelection()` to: (1) Convert logical grid coordinates to viewport-relative coordinates by subtracting `viewport.topRow` and `viewport.leftCol`, (2) Only apply highlight if cell is currently visible, (3) Store previous selection to properly remove old highlight
Formula with cell reference returns `#REF!` error	The `ParsedFormula` contains unresolved dependencies because `getCellByAddress()` returns `undefined` for valid-looking addresses. The parser fails to handle out-of-bounds references or empty cells.	1. Log the `dependencies` set extracted from the formula's AST 2. Check if `isValidCellAddress()` validates the reference format correctly 3. Verify `getCellByAddress()` handles both existing cells and returns appropriate default for empty cells	Update the value resolver in `evaluateAST()` to: (1) Return `0` (or `""`) for empty cells instead of `undefined`, (2) Validate cell addresses against `MAX_ROWS` and `MAX_COLUMNS` before attempting access, (3) Return `ERRORS.REF_ERROR` for invalid addresses
Formula `=A1+B2` calculates incorrectly (e.g., concatenates instead of adds)	**Type coercion** during evaluation treats numeric values as strings. JavaScript's `+` operator performs string concatenation when any operand is a string.	1. Log the raw values returned by `getCellByAddress()` for A1 and B2 2. Check if cell values are stored as strings (e.g., `"5"` instead of `5`) 3. Trace through `evaluateNode()` for the `BinaryOpNode` with operator `+`	Implement strict type checking in `evaluateNode()`: (1) For arithmetic operators, convert operands to numbers using `Number()` or `parseFloat()`, (2) Return `NaN` or `ERRORS.VALUE_ERROR` if conversion fails, (3) For `+` specifically, decide on numeric-only addition or implement Excel-like behavior (numbers only)
Circular reference not detected, causing infinite recalculation loop	The `detectCircularReference()` algorithm has a logic error (often missing visited node tracking) or isn't called before adding dependencies. The graph may allow self-references.	1. Add logging to `detectCircularReference()` to trace its pathfinding 2. Check if the method is called in `setCellFormula()` before updating dependencies	Implement proper cycle detection using DFS with visited set: (1) Track visited nodes in recursion, (2) Handle self-references (A1: `=A1`) as special case, (3) Clear old dependencies before adding

Symptom	Likely Cause	How to Diagnose	Fix
		3. Test with simple circular reference (A1: `=B1`, B1: `=A1`)	new ones in `setCellFormula()`
Changing one cell doesn't update dependent cells	The **dependency graph** edges weren't updated when the formula was parsed, or `getTopologicalOrder()` returns wrong order. The recalculation engine may not be triggered on value change.	1. Verify `extractDependencies()` is called and returns correct references 2. Check that `addDependency()` is called for each extracted reference 3. Log the output of `getTopologicalOrder()` for the changed cell	Ensure `handleCellEdit()`: (1) Calls `extractDependencies()` on new formula, (2) Removes old dependencies via `removeDependency()`, (3) Adds new dependencies, (4) Calls `getTopologicalOrder()` and evaluates each cell in order
Undo works once then stops	The **command pattern** implementation has incorrect stack management. The `HistoryStack` may have limited depth (`maxDepth`) or commands aren't being pushed correctly.	1. Log the `undoStack` and `redoStack` lengths after each operation 2. Check if `push()` clears the redo stack as expected 3. Verify command objects store enough state for proper inversion	Fix `HistoryStack.push()`: (1) Ensure it clears `redoStack`, (2) Trim `undoStack` when exceeding `maxDepth`, (3) Store complete cell state (value, formula, format) in command, not just value
Pasting formula adjusts references incorrectly	The `adjustFormula()` method miscalculates the **displacement vector** between source and target, or fails to handle **absolute references** (with `$`). Relative references shift when they shouldn't.	1. Log sourceAddress, targetAddress, and displacement (rowDiff, colDiff) 2. Test with formulas containing mixed references (`A1`, `$A1`, `A$1`, `$A$1`) 3. Check if `parseCellAddress()` correctly identifies absolute flags	Update `adjustFormula()` to: (1) Parse formula into AST, (2) Traverse AST and adjust only `CellReferenceNode` without absolute flags, (3) Reconstruct formula from modified AST, (4) Handle range references (`A1:B10`) by adjusting both start and end
SUM(A1:A10) includes empty cells as zero incorrectly	The range resolution in `resolveRange()` includes all cells in range, but the evaluation treats empty cells as `0`. Excel's `SUM` ignores empty cells but treats them as `0` in arithmetic.	1. Check what values `resolveRange()` returns for empty cells 2. Verify how `FunctionCallNode` for `SUM` handles `undefined` or empty values 3. Test with range containing mix of numbers, empties, and text	Decide on Excel-compatible behavior: `SUM` should ignore text and empty cells (treat as 0). Implement in `evaluateNode()` for `FunctionCallNode`: (1) Filter out non-numeric values, (2) Use `Number()` conversion with `isNaN()` check, (3) Sum only valid numbers
Memory leak: performance degrades with many formulas	The **dependency graph** retains references to deleted cells, or DOM elements in `cellPool` aren't properly managed. Event listeners may accumulate on recycled elements.	1. Monitor heap usage in browser DevTools when creating/deleting cells 2. Check if `Cell` objects are removed from `DependencyGraph.nodes` when cleared 3. Verify event listeners are removed or use **event delegation**	Implement cleanup: (1) Remove cell from `DependencyGraph.nodes` when cell is deleted, (2) Remove all edges pointing to/from deleted cell, (3) Use event delegation on grid container instead of per-cell listeners, (4) Clear cell content before recycling in `renderViewport()`

> **Key Insight:** Many spreadsheet bugs manifest as *incorrect calculations* but originate from *incorrect dependency tracking*. Always verify the dependency graph structure before debugging calculation logic itself.

Architecture Decision Record: Debug-First Development Approach

> **Decision: Implement comprehensive debug visualization hooks from the start**
>
> - **Context**: Spreadsheet bugs are notoriously difficult to trace due to the reactive dataflow and complex dependencies. Without visualization tools, developers waste time adding console logs and mentally tracing execution paths.
> - **Options Considered**:
>   1. **Add logging as needed**: Implement minimal logging and add more when bugs occur (traditional approach)
>   2. **Build debug visualization system**: Design and implement a comprehensive debug panel with graph visualization, formula tracing, and state inspection from the beginning
>   3. **Use external debugging tools**: Rely on browser DevTools and manual inspection of data structures
> - **Decision**: Build debug visualization system integrated into the application
> - **Rationale**: The upfront cost of building debug tools pays exponential dividends throughout development. Visualization helps understand the system's actual behavior versus expected behavior. For a learning project, these tools provide immediate feedback about how formulas, dependencies, and recalculation work.
> - **Consequences**: Additional code complexity for debug features, but these can be conditionally compiled or feature-flagged. Provides excellent learning aids and reduces frustration during development.

Option	Pros	Cons	Why Not Chosen
Add logging as needed	Minimal initial effort, flexible	Reactive debugging is inefficient, hard to see system state holistically	Leads to "printf debugging" which is inadequate for graph-based systems
Build debug visualization system	**Provides holistic view, accelerates debugging, educational value**	**Initial development overhead, additional UI complexity**	**Chosen for educational value and development efficiency**
Use external debugging tools	No additional code, uses proven tools	Limited to browser capabilities, can't visualize custom data structures	Browser DevTools can't visualize dependency graphs or formula ASTs effectively

Spreadsheet-Specific Debugging Techniques

Beyond generic debugging approaches, spreadsheet systems require specialized techniques to visualize and trace their unique behaviors. These methods transform the abstract dependency graph and formula evaluation into observable, inspectable representations.

1. Dependency Graph Visualization

The **dependency graph** is the most critical data structure to visualize when debugging recalculation issues. Implement these visualization techniques:

Technique A: Console Representation Create a `printGraph()` method that outputs the graph in adjacency list format to the console. This should show each cell and its direct dependencies:

```
A1: [B1, C2]       // A1 depends on B1 and C2
B1: [D3]           // B1 depends on D3
C2: []             // C2 has no dependencies
D3: [A1]           // CIRCULAR REFERENCE DETECTED!
```
PLAINTEXT

Technique B: Visual Graph Renderer For complex dependencies, implement an HTML overlay that draws nodes and edges on top of the grid. Use absolute positioning to place nodes at cell centers and draw SVG lines between them. Color-code edges based on dependency type (direct vs transitive) and highlight circular references in red.

Technique C: Impact Analysis Display When a cell is selected, visually highlight all cells that depend on it (forward reachability) and all cells it depends on (backward reachability). Use semi-transparent colored overlays—green for dependencies, orange for dependents. This immediately shows the "blast radius" of a cell change.

2. Formula Evaluation Tracing

Formula evaluation involves multiple stages: tokenization, parsing, AST construction, and evaluation. Implement tracing for each stage:

Tracing Level	What to Display	Diagnostic Value
Tokenization	List of tokens with type and value: `[NUMBER: "5"]`, `[OPERATOR: "+"]`, `[CELL_REF: "A1"]`	Reveals parsing errors at lexical level (e.g., malformed cell references)
AST Construction	Tree structure showing node hierarchy and types (indented text or visual tree)	Shows operator precedence issues and expression grouping
Evaluation Path	Step-by-step evaluation with intermediate values: `5 + A1 → 5 + 10 → 15`	Identifies incorrect operator behavior or type coercion
Value Resolution	Cell reference → value mapping: `A1 → 10, B2 → "text", C3 → undefined`	Shows whether cell values are being fetched correctly

Implementation Approach: Add a `trace` option to `parseFormula()` and `evaluateAST()` that, when enabled, logs each step to a dedicated debug panel or console. Use indentation to show recursion depth.

3. Recalculation Flow Monitoring

The recalculation engine's behavior during cell updates is critical to debug. Implement these monitoring techniques:

Step-by-Step Recalculation Playback When a cell changes, record: (1) The initial changed cell, (2) The topological order determined by `getTopologicalOrder()`, (3) Each cell evaluation in order with before/after values. Present this as a timeline that can be stepped through like a debugger.

Dirty Flag Visualization Cells that need recalculation are "dirty." Implement visual indicators (small red dot in cell corner) showing dirty state. This helps verify that only necessary cells are marked for recalculation and that the dirty state clears after recalculation.

Performance Profiling Hooks Instrument the recalculation engine to track: (1) Number of cells recalculated per change, (2) Time spent in topological sort vs evaluation, (3) Cache hit rates for formula parsing. Display as real-time metrics to identify performance bottlenecks.

4. State Inspection Tools

Cell State Inspector Implement a panel that shows complete cell state when a cell is clicked: raw value, formula string, format object, dependencies list, and calculated value. Allow editing these properties directly to test hypotheses.

History Stack Visualizer Display the undo/redo stacks as stacks of cards showing command descriptions. Allow clicking on any command to preview the spreadsheet state at that point in history without actually executing undo/redo.

Viewport Debug Overlay Show the current viewport boundaries (topRow, leftCol, height, width) and which logical cells map to which DOM elements in the cell pool. This helps diagnose virtual scrolling issues.

5. Test Scenario Recorder and Replayer

Create a tool that records user interactions (cell edits, selections, scrolls) and can replay them exactly. This is invaluable for reproducing intermittent bugs and verifying fixes don't regress existing behavior.

Recording Format Example:

```json
{
  "actions": [
    {"type": "setSelection", "row": 0, "col": 0, "timestamp": 1000},
    {"type": "setCellFormula", "address": "A1", "formula": "=B1+5", "timestamp": 1500},
    {"type": "setCellValue", "address": "B1", "value": 10, "timestamp": 2000}
  ]
}
```

6. Common Pattern Detection

Implement automatic detection of common bug patterns:

Pattern	Detection Method	Common Bug
Unintended Circular Reference	Run `detectCircularReference()` on every dependency addition and log warnings	Missing cycle detection in graph updates
Orphaned Dependencies	Find cells in dependency graph that don't exist in spreadsheet grid	Failed cleanup when deleting cells
Type Coercion Surprises	Log all type conversions during evaluation with source values	String/number confusion in formulas
Viewport Mismatch	Compare visible DOM cells with expected viewport cells	Incorrect scroll offset calculations

> **Key Insight:** The most effective spreadsheet debugging technique is making the invisible visible. Dependency graphs, formula ASTs, and recalculation order are normally hidden—exposing them through visualization transforms debugging from guesswork to systematic investigation.

Implementation Guidance

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
Debug Visualization	Console logging with formatted strings	Dedicated debug panel using HTML/SVG with D3.js for graph visualization
Formula Tracing	`console.group()` with indentation for AST traversal	Interactive AST visualizer with collapsible nodes and hover details
Performance Monitoring	`console.time()` / `console.timeEnd()` for critical sections	Real-time metrics dashboard using Chart.js or similar
State Inspection	Modal dialog showing JSON representation of cell state	Resizable inspector panel with editable fields and live preview

B. Recommended File/Module Structure

Add debug components to the project structure:

```
project-root/
  src/
    components/
      grid/                    # Existing grid components
      debug/                   # New debug components
        DebugPanel.js          # Main debug panel UI component
        GraphVisualizer.js     # Dependency graph visualization
        FormulaTracer.js       # Formula evaluation tracing display
        StateInspector.js      # Cell state inspection tool
    utils/
      debug/                   # Debug utilities
        logger.js              # Enhanced logging with levels
        performance.js         # Performance measurement utilities
        recorder.js            # Action recorder for test replay
```

C. Infrastructure Starter Code

Create a comprehensive debug logger that can be enabled/disabled:

JAVASCRIPT

```javascript
// src/utils/debug/logger.js

/**
 * Debug logger with configurable levels and module-based filtering
 */
export class DebugLogger {
  constructor() {
    this.levels = {
      ERROR: 0,
      WARN: 1,
      INFO: 2,
      DEBUG: 3,
      TRACE: 4
    };

    this.currentLevel = this.levels.INFO;
    this.enabledModules = new Set(['graph', 'parser', 'recalc']);
  }

  /**
   * Configure logging for specific modules
   * @param {Object} config - {level: 'DEBUG', modules: ['graph', 'parser']}
   */
  configure(config) {
    if (config.level && this.levels[config.level] !== undefined) {
      this.currentLevel = this.levels[config.level];
    }
    if (config.modules) {
      this.enabledModules = new Set(config.modules);
    }
  }

  /**
```

```
   * Log message if module is enabled and level sufficient
   * @param {string} module - Module name ('grid', 'parser', 'graph', etc.)
   * @param {string} level - Log level ('DEBUG', 'INFO', etc.)
   * @param {string} message - Log message
   * @param {any} data - Additional data to log
   */
  log(module, level, message, data = null) {
    const levelValue = this.levels[level];

    // Check if we should log this message
    if (levelValue <= this.currentLevel && this.enabledModules.has(module)) {
      const timestamp = new Date().toISOString().split('T')[1].slice(0, -1);
      const prefix = `[${timestamp}] [${module}] [${level}]`;

      if (data !== null) {
        console.log(`${prefix} ${message}`, data);
      } else {
        console.log(`${prefix} ${message}`);
      }
    }
  }

  /**
   * Start a collapsible group for tracing
   * @param {string} module - Module name
   * @param {string} title - Group title
   */
  startGroup(module, title) {
    if (this.enabledModules.has(module) && this.currentLevel >= this.levels.DEBUG) {
      console.groupCollapsed(`[${module}] ${title}`);
    }
  }
```

```
  /**
   * End a collapsible group
   * @param {string} module - Module name
   */
  endGroup(module) {
    if (this.enabledModules.has(module) && this.currentLevel >= this.levels.DEBUG) {
      console.groupEnd();
    }
  }
}

// Singleton instance
export const debugLogger = new DebugLogger();

// Convenience methods for common modules
export const graphLogger = {
  info: (msg, data) => debugLogger.log('graph', 'INFO', msg, data),
  debug: (msg, data) => debugLogger.log('graph', 'DEBUG', msg, data),
  trace: (msg, data) => debugLogger.log('graph', 'TRACE', msg, data)
};

export const parserLogger = {
  info: (msg, data) => debugLogger.log('parser', 'INFO', msg, data),
  debug: (msg, data) => debugLogger.log('parser', 'DEBUG', msg, data)
};
```

D. Core Logic Skeleton Code

Implement a debug visualization method for the dependency graph:

JAVASCRIPT

```javascript
// src/utils/debug/graphDebug.js

/**
 * Generate a visual representation of the dependency graph for debugging
 * @param {DependencyGraph} graph - The dependency graph to visualize
 * @param {string} highlightCell - Optional cell address to highlight in visualization
 * @returns {string} Textual representation of the graph
 */
export function visualizeGraph(graph, highlightCell = null) {
  // TODO 1: Initialize output string with header
  // TODO 2: Iterate through all nodes in graph.nodes
  // TODO 3: For each cell address, get its dependencies from graph.edges
  // TODO 4: Format as "A1: [B2, C3]" with highlighting if address matches highlightCell
  // TODO 5: Detect and flag circular references (cells that appear in their own dependency chain)
  // TODO 6: Return the complete string representation
}

/**
 * Trace formula evaluation step-by-step
 * @param {string} formula - The formula to trace
 * @param {string} contextCellAddress - The cell containing the formula
 * @param {Function} valueResolver - Function to resolve cell references to values
 * @returns {Array} Array of evaluation steps with descriptions
 */
export function traceFormulaEvaluation(formula, contextCellAddress, valueResolver) {
  // TODO 1: Tokenize the formula and record tokens
  // TODO 2: Parse tokens into AST and record AST structure
  // TODO 3: Initialize steps array to collect evaluation trace
  // TODO 4: Create a wrapper for evaluateAST that records each node evaluation
  // TODO 5: Evaluate AST with tracing wrapper
  // TODO 6: Return array of steps: [{step: 1, description: "Tokenized: 5, +, A1"}, ...]
}
```

E. Language-Specific Hints

- **Console Grouping**: Use `console.groupCollapsed()` and `console.groupEnd()` to create collapsible sections in browser DevTools for different debugging modules.
- **Performance Measurement**: Use `performance.now()` for high-resolution timing instead of `Date.now()`.
- **Conditional Debugging**: Use `process.env.NODE_ENV` (in build tools) or a global flag to strip debug code in production:

```javascript
if (typeof DEBUG !== 'undefined' && DEBUG) {
  // Debug-only code
}
```

- **Error Objects**: Create custom error types for spreadsheet errors (`CircularReferenceError`, `ParseError`) that include additional context like cell address and formula.

F. Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
Formula shows `#VALUE!` but cells seem correct	Type mismatch during function evaluation (e.g., `SUM` with text values)	Enable formula tracing to see each evaluation step and intermediate values	Implement type checking in function evaluation; convert strings to numbers where appropriate
Undo causes infinite loop	Command execution triggers another command that gets pushed to undo stack	Log command execution and stack pushes; check for recursive `push()` calls	Ensure command execution doesn't trigger events that create new commands
Graph visualization shows disconnected components	Dependency extraction misses some references	Compare `extractDependencies()` output with formula string manually	Debug `extractDependencies()` with test formulas containing ranges and nested functions

G. Milestone Checkpoint

After implementing debugging tools, verify they work:

1. **Enable debug logging** in the browser console:

```javascript
// In browser DevTools console
window.debugLogger.configure({level: 'DEBUG', modules: ['graph', 'parser']});
```

2. **Test graph visualization** by creating a simple dependency chain:
   - Set A1 formula: `=B1+C1`
   - Set B1 formula: `=D1`
   - Check console for graph output showing A1→[B1,C1], B1→[D1]
3. **Verify formula tracing** by entering a complex formula:
   - Set E1 formula: `=SUM(A1:C1)*2`
   - Look for trace output showing tokenization, AST, and step-by-step evaluation
4. **Test circular reference detection**:
   - Set F1 formula: `=F1` (self-reference)

- Should see warning in console and cell should show `#CIRCULAR_REF!`

Expected console output should show clear, organized debugging information. If logs are missing or confusing, check that debug modules are properly enabled and log statements are placed at key decision points in the code.

Future Extensions

> **Milestone(s):** This section provides forward-looking guidance for potential enhancements beyond the four core milestones. It demonstrates how the current architecture enables advanced functionality and outlines design considerations for future development.

The architecture established in the previous sections was intentionally designed with **separation of concerns** and **modularity** as core principles. This foundation not only supports the current feature set but also creates a robust platform for future enhancements. The **reactive dataflow** model, the **dependency graph** engine, and the **virtualized grid** rendering provide extensible interfaces and patterns that can accommodate significant new functionality without major architectural refactoring. This section explores several high-value extensions, detailing how they would integrate with the existing system, the design accommodations already in place, and the modifications required for implementation.

Extension Ideas and Design Accommodations

The following extensions represent natural evolutions of the spreadsheet application, each building upon the established architectural layers. For each extension, we examine the core concept, the required architectural changes, and the specific design decisions that either enable or must be adapted for its implementation.

Multi-Sheet Support

Mental Model: A spreadsheet **workbook** containing multiple **worksheets**, akin to a binder with many tabbed pages. Each sheet is an independent `Spreadsheet` data model instance, but formulas can reference cells across different sheets, creating a network of inter-sheet dependencies.

Architectural Impact and Design Accommodations: The current architecture models a single sheet. Multi-sheet support requires extending the data model to manage a collection of sheets and enhancing the formula parser and dependency graph to handle cross-sheet references (e.g., `Sheet2!A1` or `Expenses!B5:B10`).

Component	Changes Required	Design Notes
Data Model	Introduce a `Workbook` type containing a map of `Sheet` objects (each a `Spreadsheet`). The `Cell` model remains unchanged, but cell addresses become three-part: `sheetName!columnRow`.	The `Spreadsheet` type becomes a representation of a single sheet. The `Workbook` becomes the root state container.
Formula Parser	Extend `parseCellAddress` and `TokenType` to recognize and parse sheet name prefixes and the delimiter (`!`). The `ParsedCellReference` type gains a `sheetName: string | null` field.	References without a sheet prefix default to the current sheet context. Absolute sheet references (e.g., `'Sheet1'!$A$1`) may also be supported.
Dependency Graph	The `DependencyGraph` must be scoped per `Workbook`. Node identifiers must include the sheet name (e.g., `"Sheet1!A1"`). `getTopologicalOrder` must traverse dependencies across sheets.	The graph's adjacency lists now connect nodes across different sheet sub-graphs, but the topological sort algorithm remains identical.
Virtual Grid UI	Add a tab bar UI component to switch between sheets. The `VirtualGrid` component's `spreadsheet` prop would switch to the active sheet's data. Viewport state (`topRow` , `leftCol`) could be stored per sheet.	The `VirtualGrid` only renders the active sheet. Efficient memory use for many sheets may require lazy-loading sheet data models.

Decision: Three-Part Cell Addressing for Multi-Sheet Formulas

- **Context**: To support formulas that reference cells across multiple worksheets, a clear and unambiguous addressing syntax is required.
- **Options Considered**:
  1. **Excel-style `SheetName!A1`** : Familiar to users, explicit delimiter.
  2. **Implicit context-based referencing**: References without a prefix search upward through named ranges or a defined hierarchy. More complex to implement and potentially ambiguous.
  3. **Unique global cell IDs (UUIDs)**: Decouples reference from location, enabling robust cell moving. Very unfamiliar to users and cumbersome for manual formula entry.
- **Decision**: Adopt the Excel-style `SheetName!A1` syntax with an optional sheet name (defaulting to current sheet).
- **Rationale**: Maximizes user familiarity and reduces learning friction. The delimiter provides a clear lexical token for the parser. It also aligns with the industry standard, making import/export to other applications more straightforward.
- **Consequences**: The parser tokenizer must recognize the `!` character as a distinct token. The `ParsedCellReference` object must store an optional sheet identifier. Absolute sheet references (`'Sheet Name'!$A$1`) require additional syntax (quotes for sheets with spaces).

Collaborative Editing (Real-Time)

Mental Model: A **shared digital whiteboard** where multiple users edit the same spreadsheet simultaneously, with changes reflected in near real-time for all participants. This transforms the application from a single-user tool to a multi-user collaborative workspace.

Architectural Impact and Design Accommodations: Real-time collaboration requires a client-server architecture with conflict resolution strategies (Operational Transformation or Conflict-Free Replicated Data Types). The current client-side state management and command pattern for undo/redo provide a solid foundation for representing local edits as operations.

Architectural Layer	Changes Required	Design Notes
Network & Sync	Introduce a `CollaborationService` client component that connects to a backend WebSocket server. Local commands (`SetCellValueCommand` , `SetCellFormulaCommand`) are serialized and broadcast as operations.	Operations must be transformed to ensure convergence (all users see the same final state). The `CompoundCommand` is useful for batching a user's single action (e.g., paste).
Data Model & State	The `Spreadsheet` becomes a **shared data model**. Each operation must be applied deterministically. The local `HistoryStack` may shift to track local undo relative to a shared baseline version.	Consider representing cell state with a version vector or last-writer-wins timestamp to resolve simultaneous edits to the same cell.
UI & Conflict Resolution	The `VirtualGrid` must visualize other users' selections/cursors (via decorations). Edit conflicts require a resolution policy (e.g., last write wins, with notification).	The `AdvancedFeaturesManager` must coordinate with the `CollaborationService` so that copy/paste operations are transmitted as a batch.

Decision: Operational Transformation (OT) over Conflict-Free Replicated Data Types (CRDTs) for Cell Grid

- **Context**: The spreadsheet grid is a large, sparse 2D array where operations (cell value updates, formula changes) are discrete and often location-based (insert/delete row could be a future extension). We need a consistency algorithm that ensures all users see the same cell values.
- **Options Considered**:
  1. **Operational Transformation (OT)**: Server maintains canonical state and transforms incoming operations against concurrent edits before applying. Well-suited for text documents and ordered sequences.
  2. **CRDTs**: Each client maintains a replicable data structure that automatically converges mathematically. Excellent for eventually consistent scenarios, but designing a CRDT for a spreadsheet grid with formulas and dependencies is complex.
  3. **Centralized Authority with Locking**: The server serializes all edits, granting temporary locks for cell ranges. Simple but harms responsiveness and concurrency.
- **Decision**: Implement OT for cell value/formula updates, leveraging existing command objects as operations.
- **Rationale**: The command pattern already encapsulates discrete state changes. OT is a proven model for collaborative editors (Google Docs). For the core use case of editing distinct cells, transformation is often straightforward (independent operations commute). A central server can also enforce validation (circular reference checks) on the transformed operation sequence.
- **Consequences**: Requires implementing a transformation function for each command type (e.g., how does `setCellValue("A1", 5)` transform against a concurrent `setCellFormula("A1", "=B1")` ?). The server becomes stateful, managing the transformation log. Undo/redo becomes more complex, as it must undo in the context of the transformed operation history.

Expanded Built-in Function Library

Mental Model: Adding more **specialized calculation tools** to the user's toolbox, moving from basic arithmetic and `SUM` to statistical, financial, logical, and text manipulation functions.

Architectural Impact and Design Accommodations: The `FormulaParser` and `AST` evaluation system are designed to be extended. New functions are primarily a matter of registration in a function registry and implementation of their evaluation logic.

Component	Changes Required	Design Notes
Parser	The tokenizer already recognizes function names. The `FunctionCallNode` in the AST is generic.	No structural changes needed. The parser's grammar must accommodate any new function syntax (e.g., variable arguments).
Evaluator	Extend the `valueResolver` or create a `FunctionRegistry` that maps function names to implementation functions. Each implementation receives evaluated argument arrays and returns a result or error.	Functions should be pure (no side effects) to maintain deterministic recalculation. Error propagation (`#VALUE!` , `#DIV/0!`) must be consistent.
Dependency Analysis	Some functions have special dependency behaviors. For example, `OFFSET(A1,1,1)` creates a dependency on `A1` but also on the dynamic offset result cell? Our system treats `OFFSET` as a regular function, making its result depend only on its explicit arguments (A1, 1, 1).	To support dynamic ranges like `OFFSET` , the `extractDependencies` function must evaluate the function's semantics or the function must be treated as a primitive during dependency extraction.

Example Function Categories and Implementation Considerations:

Category	Example Functions	Implementation Notes
Logical	`IF` , `AND` , `OR` , `NOT`	`IF` requires lazy evaluation of branches (only evaluate the chosen branch) to avoid errors in the unused branch. This may require special handling in `evaluateAST` .
Text	`CONCAT` , `LEFT` , `FIND` , `LEN`	Ensure proper string/number coercion. Functions returning numbers can be used in arithmetic.
Lookup & Reference	`VLOOKUP` , `HLOOKUP` , `INDEX` , `MATCH`	These functions introduce dependencies on entire table ranges. Performance becomes critical for large ranges.
Statistical	`STDEV` , `VAR` , `MEDIAN`	May require sorting or full array traversal. Consider efficiency for large ranges.
Financial	`PMT` , `FV` , `NPV`	Require floating-point precision awareness.

Decision: Centralized Pure Function Registry for Evaluation

- **Context**: As the library grows from 5 to 50+ functions, managing their implementation and ensuring consistent error handling becomes crucial.
- **Options Considered**:
  1. **Monolithic evaluate function with switch statement**: All function logic in one large `evaluateFunctionCall` switch. Simple for few functions, but becomes unwieldy.
  2. **Decentralized function registry**: A map from function name to a function implementation object. Each implementation is responsible for argument validation, evaluation, and error handling.
  3. **Plugin architecture**: Functions are defined in separate modules that register themselves. Enables third-party extensions but adds complexity.
- **Decision**: Implement a centralized `FunctionRegistry` using a map, with each function defined as a separate module for maintainability.
- **Rationale**: The registry pattern provides a clean separation of concerns, makes testing individual functions easier, and allows for dynamic addition/removal of functions (useful for potential user-defined functions). Keeping functions as pure, stateless implementations aligns with the reactive dataflow model.
- **Consequences**: The registry must be initialized before parsing/evaluation. Function names are case-insensitive (typically). Argument arity and type checking must be performed consistently across all functions.

Charting and Data Visualization

Mental Model: Graphical representations of cell data that update automatically when source data changes—a live dashboard linked to the spreadsheet's reactive engine.

Architectural Impact and Design Accommodations: Charts are a new presentation layer component that observe specific cell ranges in the data model. They must subscribe to changes in those ranges and re-render accordingly.

Component	Changes Required	Design Notes
Data Model	Extend `Spreadsheet` or `Workbook` to contain a collection of `Chart` objects. Each `Chart` defines a type (line, bar, pie), a data range (e.g., `"A1:B10"`), and display properties.	Charts are part of the persistent spreadsheet state, saved/loaded alongside cell data.
Dependency System	A `Chart` must be registered as a dependent of its data range cells. When any cell in the range changes, the chart must be flagged for re-rendering.	This can be implemented by creating a special "chart node" in the `DependencyGraph` that depends on the range cells. Recalculation for charts means triggering a redraw, not a value recomputation.
Rendering Engine	A new `ChartRenderer` component using a canvas or SVG library (e.g., D3.js, Chart.js). It listens to the chart's data range changes and updates the visual.	Charts may be rendered in a separate pane or floating overlays on the grid. The `VirtualGrid` may need to allocate space for inline charts.

Integration Flow:

1. User selects a cell range and clicks "Insert Chart".
2. A `CreateChartCommand` is executed, adding a `Chart` object to the workbook and adding dependency edges from the chart to each cell in the range.
3. The `DependencyGraph`'s `getDependents` for any cell in the range now includes the chart ID.
4. When a source cell changes, the topological sort includes the chart ID in the recalculation order.
5. A chart-specific "recalculation" handler is invoked, which triggers the `ChartRenderer` to fetch the new range data and redraw.

Decision: Treat Charts as Special Formula Cells with Rendering Side Effects

- **Context**: Charts depend on cell data and must update when that data changes, similar to formula cells. However, their output is a visual rendering, not a cell value.
- **Options Considered**:
  1. **Charts as graph dependents**: Extend the dependency graph to include chart nodes. When source cells recalculate, mark charts as dirty and trigger a render.
  2. **Observer pattern on cell ranges**: Charts subscribe to a cell range observable outside the dependency graph. Simpler but duplicates the dependency tracking mechanism.
  3. **Manual refresh**: Charts only update on user command or periodic timer. Does not provide live updates.
- **Decision**: Integrate charts into the dependency graph as special nodes.
- **Rationale**: Leverages the existing, robust dependency and recalculation engine. Ensures charts update atomically with all other dependents in the correct order (e.g., after formulas that compute the chart data). Provides a consistent model for serialization and undo/redo of chart creation/deletion.
- **Consequences**: The `DependencyGraph` must support node types beyond cells. The recalculation engine must know how to "evaluate" a chart node (i.e., queue it for rendering). This creates a clean separation: the dataflow engine manages *when* charts update, and the rendering component manages *how*.

Performance Optimizations

Mental Model: Making the spreadsheet **feel instantaneous** even with thousands of complex formulas, by avoiding unnecessary work and intelligently scheduling computations.

The current architecture already includes foundational optimizations (virtualized rendering, topological sort for minimal recalculation). Further optimizations target the recalculation engine and graph operations.

Optimization	Concept	Implementation Approach
Incremental Topological Sort	Instead of re-sorting the entire dependency subgraph for each change, maintain a partial order and update it incrementally.	Implement the **Dynamic Topological Sort** algorithm. When an edge is added or removed, update the order locally rather than recomputing from scratch. This is complex but beneficial for very large graphs with frequent small changes.
Lazy Evaluation & Dirty Flags	Only evaluate a formula when its value is needed (e.g., for display or for another formula that is being evaluated).	Each `Cell` already has a `dirty` flag (implicitly, by being in the set of cells returned by `getTopologicalOrder`). We can extend this by skipping evaluation of dirty cells that are not in the viewport and not dependencies of any visible cell. Requires tracking "viewed" cells.
Memoization of Formula Results	Cache the evaluated result of a formula as long as its dependencies haven't changed.	The `Cell`'s `value` field serves as this cache. The recalculation engine already updates it when dependencies change. For complex functions, internal function evaluation could also be memoized.
Parallel Recalculation	Evaluate independent branches of the dependency graph simultaneously using web workers.	The topological order provides a sequence, but independent cells at the same "level" can be evaluated in parallel. The graph can be analyzed for independent sets. Web Workers introduce communication overhead; best for heavy computational functions.
Sparse Graph Storage Optimizations	The adjacency lists in `DependencyGraph` may become large. Use more efficient data structures.	Use numeric cell IDs (row/column packed into a number) instead of string addresses as map keys to reduce memory. Use `Set` for dependents, but consider switching to arrays if iteration performance is critical.

Decision: Prioritize Simplicity Over Advanced Optimizations Initially

- **Context**: The initial implementation focuses on correctness and clarity. Premature optimization can complicate the codebase and introduce bugs.
- **Options Considered**:
  1. **Implement all optimizations from the start**: Build with incremental topological sort, lazy evaluation, and parallel workers.
  2. **Implement a straightforward, correct engine first**: Use full topological sort on the affected subgraph, evaluate all dirty cells, and optimize only when performance becomes a measurable issue.
  3. **Build with optimization hooks**: Design interfaces that allow optimizations to be added later without changing the core API.
- **Decision**: Start with a straightforward engine (as described in Milestone 3) but design the `DependencyGraph` interface to be implementation-agnostic, allowing for a more optimized engine to be swapped in later.
- **Rationale**: The educational value of understanding the clear, canonical algorithms (topological sort) outweighs the performance benefit for typical project spreadsheets (hundreds of cells). A simple implementation is easier to debug and verify. The interface-based design (`DependencyGraph` as an abstract type) allows for later replacement with an `OptimizedDependencyGraph` that has the same methods but different internals.
- **Consequences**: Large spreadsheets (10,000+ cells with deep dependency chains) may experience noticeable lag on recalculation. This is an acceptable trade-off for the learning project. If performance becomes critical, the optimization work is isolated to a single component.

Additional Extensions Brief

- **Conditional Formatting:** Extend the `CellFormat` model to include rules (e.g., `if value > 100 then background=red`). The recalculation engine must re-apply formatting rules when cell values change. This can be implemented by treating formatted cells as dependents of the rule's condition formula.
- **Named Ranges:** Allow users to assign a name (e.g., `"SalesTax"`) to a cell or range. The parser would resolve these names. Requires a global name registry in the `Workbook` and an update to the formula tokenization and reference resolution.
- **Data Validation:** Attach validation rules (e.g., "value must be between 1 and 10") to cells. The `AdvancedFeaturesManager` or a new `ValidationManager` would check values on edit and provide feedback.
- **Scripting/Macros:** Allow user-defined functions in JavaScript. This would expose a safe sandbox for executing user code, which could call into the spreadsheet API to read/write cells. Major security considerations are required.
- **Import/Export for Other Formats (XLSX, ODS):** Leverage existing libraries (e.g., SheetJS) to read/write industry-standard file formats. The `AdvancedFeaturesManager`'s import/export functions would be extended to delegate to these libraries after mapping to/from the internal data model.

Each of these extensions demonstrates the flexibility of the layered architecture. The **data model** can be enriched with new properties, the **dependency graph** can incorporate new node types, the **formula parser** can be extended with new syntax, and the **UI components** can be added or enhanced without disrupting the core recalculation engine.

Implementation Guidance

While the Future Extensions are not part of the immediate implementation milestones, setting up the codebase with extensibility in mind will make these enhancements much easier. Below are technology recommendations and architectural accommodations to consider during the initial build.

A. Technology Recommendations Table

Component	Simple Option (Initial Build)	Advanced Option (For Extensions)
Multi-Sheet State Management	Store sheets in a plain object `{ [name]: Spreadsheet }` within the `Workbook`.	Use a state management library (Redux, Zustand) with normalized state for efficient updates and persistence.
Collaborative Editing	Use a simple WebSocket server with operational transformation library (e.g., `sharedb` or `yjs`).	Implement a custom OT server with Redis for scaling and persistence, using the Command objects as operations.
Chart Rendering	Use a lightweight charting library like `Chart.js` for simple chart types.	Use a more powerful, low-level library like `D3.js` for fully customizable visualizations.
Performance Optimizations	Use the basic topological sort; optimize only if performance issues arise.	Implement incremental topological sort using a library like `graphlib` or a custom dynamic graph data structure.
Additional Functions	Implement new functions directly in the `evaluateFunctionCall` switch.	Create a plugin system where functions are registered via a configuration object, allowing easy addition.

B. Recommended File/Module Structure for Extensibility

To accommodate future extensions, organize the codebase with clear separation and placeholder directories:

```
project-root/
  src/
    core/
      data-model/          # Cell, Spreadsheet, Workbook (future)
        cell.ts
        spreadsheet.ts
        workbook.ts        # Future: multi-sheet support
      formula/
        parser.ts
        ast.ts
        evaluator.ts
        functions/         # Built-in function implementations
          sum.ts
          average.ts
          financial.ts     # Future: PMT, FV
          statistical.ts   # Future: STDEV, MEDIAN
      graph/
        dependency-graph.ts
        topological-sort.ts
    ui/
      virtual-grid/
        VirtualGrid.tsx    # Main grid component
        ScrollManager.ts
        CellRenderer.tsx
      components/
        FormulaBar.tsx
        SheetTabs.tsx      # Future: multi-sheet UI
        ChartContainer.tsx # Future: chart rendering
    features/
      advanced/
        undo-redo.ts
        clipboard.ts
        csv-export.ts
        formatting.ts
      collaboration/       # Future: real-time sync
        CollaborationService.ts
        operation-transforms.ts
      charts/              # Future: charting engine
        ChartModel.ts
        ChartRenderer.ts
    utils/
      cell-address.ts
      csv-parser.ts
```

C. Infrastructure Starter Code for Extensibility Hooks

Even if not implementing extensions immediately, you can create placeholder interfaces and factory methods to make later integration smoother.

1. Function Registry Skeleton (for adding new functions):

```typescript
// src/core/formula/functions/registry.ts

type FunctionImpl = (args: any[], context: EvaluationContext) => any;


export class FunctionRegistry {

  private functions: Map<string, FunctionImpl> = new Map();


  register(name: string, impl: FunctionImpl): void {

    this.functions.set(name.toLowerCase(), impl);

  }


  get(name: string): FunctionImpl | undefined {

    return this.functions.get(name.toLowerCase());

  }


  // Initialize with built-in functions

  static createDefault(): FunctionRegistry {

    const registry = new FunctionRegistry();

    registry.register('SUM', sumImplementation);

    registry.register('AVG', averageImplementation);

    // ... register others

    return registry;

  }

}


// Usage in evaluator:

// const registry = FunctionRegistry.createDefault();

// const fn = registry.get(node.functionName);

// if (fn) { result = fn(evaluatedArgs, context); }
```

2. Chart Model Placeholder (for future charting):

```typescript
// src/features/charts/ChartModel.ts (future)

export interface Chart {
  id: string;
  type: 'line' | 'bar' | 'pie';
  dataRange: string; // e.g., "Sheet1!A1:B10"
  title?: string;
  // ... other properties
}

export class ChartManager {
  private charts: Map<string, Chart> = new Map();

  addChart(chart: Chart): void {
    this.charts.set(chart.id, chart);
    // TODO: Register chart as dependent of cells in dataRange
  }

  updateChartData(chartId: string): void {
    const chart = this.charts.get(chartId);
    if (!chart) return;
    // TODO: Fetch current data from spreadsheet and notify renderer
  }
}
```

TYPESCRIPT

D. Core Logic Skeleton for a New Feature (Example: Multi-Sheet Reference Parsing)

When implementing multi-sheet support, the `parseCellAddress` function will need enhancement. Here is a skeleton with TODOs:

TYPESCRIPT

```typescript
// src/utils/cell-address.ts

export interface ParsedCellReference {
  sheetName: string | null;  // NEW field for multi-sheet
  row: number;
  column: number;
  rowAbsolute: boolean;
  columnAbsolute: boolean;
}

export function parseCellAddress(address: string): ParsedCellReference {
  // TODO 1: Check for '!' delimiter in the address string
  // TODO 2: If '!' is present, split into sheetNamePart and cellAddressPart
  // TODO 3: Remove optional single quotes from sheetNamePart if present (for sheets with spaces)
  // TODO 4: Parse the cellAddressPart using the existing logic for column/row/absolute markers
  // TODO 5: Return a ParsedCellReference object with the sheetName field populated (or null if no sheet)
  // TODO 6: Validate that sheetName is not an empty string after trimming quotes
  // Hint: Use regular expression or iterative character scanning
}

export function toCellAddress(
  row: number,
  column: number,
  rowAbsolute: boolean = false,
  columnAbsolute: boolean = false,
  sheetName: string | null = null  // NEW optional parameter
): string {
  // TODO 1: Generate the cell address part using columnToLetter and row number, with $ for absolute
  // TODO 2: If sheetName is provided, prepend it with proper quoting if needed (e.g., contains spaces)
  // TODO 3: If sheetName contains spaces or special characters, wrap it in single quotes
  // TODO 4: Combine sheetName and cell address with '!' delimiter
  // TODO 5: Return the full address string
}
```

E. Language-Specific Hints for JavaScript/TypeScript

- **Web Workers for Parallel Recalculation**: Use `new Worker('./calc-worker.js')` to offload heavy formula evaluation. Pass cell data and formulas as serializable objects. Note that functions cannot be serialized; the worker must have its own copy of the evaluation logic.
- **Observable Pattern for Chart Updates**: Use RxJS or a simple custom observable to allow charts to subscribe to cell range changes. Alternatively, use the existing dependency graph to notify chart components.
- **Operational Transformation Libraries**: Consider using `sharedb` or `yjs` for collaborative editing. `yjs` provides CRDTs and is well-suited for real-time collaborative applications without a central server.
- **Dynamic Import for Function Plugins**: Use `import()` to lazy-load function implementations only when they are used in a formula, reducing initial bundle size.

F. Milestone Checkpoint for Future Work

While there are no formal milestones for these extensions, a good checkpoint after completing the core four milestones is to implement one extension fully:

Checkpoint: Implement Multi-Sheet Support

1. **Extend Data Model**: Create a `Workbook` class that holds multiple `Spreadsheet` instances.
2. **Update Parser and Address Utilities**: Modify `parseCellAddress` and `toCellAddress` to handle sheet names.
3. **Update Dependency Graph**: Ensure node IDs include sheet prefix.
4. **Add UI Tabs**: Implement a tab bar to switch between sheets.
5. **Test**: Create a test with two sheets where a formula in `Sheet2` references a cell in `Sheet1`. Verify that changing the source cell triggers recalculation in the dependent sheet.

Expected Behavior:

- Opening the app shows a tab bar with "Sheet1".
- User can add a new sheet via a button.
- Formula `=Sheet1!A1 + 5` in `Sheet2!B2` correctly displays the sum.
- Changing the value in `Sheet1!A1` updates the result in `Sheet2!B2`.

G. Debugging Tips for Extension Implementation

Symptom	Likely Cause	How to Diagnose	Fix
Cross-sheet formula shows `#REF!`	The dependency graph is not connecting nodes across sheets.	Log the node IDs being added to the graph. Check if sheet prefix is included.	Ensure `addDependency` uses the full address with sheet name.
Chart does not update when data changes	Chart is not registered as a dependent of the data range cells.	Use `visualizeGraph` to see if chart node appears and has edges from source cells.	In chart creation, explicitly add edges from chart node to each cell in the range.
New function causes parser error	Function name tokenization is failing or function is not in registry.	Check token stream for the function name. Verify the function is registered before parsing.	Add function to the `FunctionRegistry` before parsing formulas that use it.
Collaborative editing causes cells to diverge	Operational transformation is not correctly transforming concurrent operations.	Log operations and their transformation on server and client. Check if timestamps or vector clocks are synchronized.	Ensure transformation functions are commutative and associative for independent operations. Use a tested OT library.

Glossary

> **Milestone(s):** All (provides reference definitions for terminology used throughout all four milestones)

Term Definitions and References

The spreadsheet system uses specialized terminology from multiple domains: graph theory, parsing theory, user interface design, and spreadsheet-specific concepts. This glossary provides clear definitions for these terms as used throughout the design document.

Term	Definition	Context
Abstract Syntax Tree (AST)	A hierarchical tree representation of the grammatical structure of a formula, where each node corresponds to a syntactic element (operator, function call, reference, etc.). The `ASTNode` union type represents the various node types in our system.	Formula parsing and evaluation
Absolute Reference	A cell reference that remains fixed when copied to a new location, denoted by dollar signs (e.g., `$A$1`). The `ParsedCellReference.rowAbsolute` and `columnAbsolute` fields track this property.	Copy/paste operations, formula adjustment
Adjacency List	A graph representation where each node stores a list of its adjacent nodes. In our `DependencyGraph` , the `edges` Map contains adjacency lists mapping from dependent cells to the cells they depend on.	Dependency graph implementation
ASTNode	A TypeScript union type representing any node in the abstract syntax tree, with specific variants for different syntactic elements: `NumberNode` , `CellReferenceNode` , `BinaryOpNode` , `UnaryOpNode` , `FunctionCallNode` , `RangeNode` , and `ErrorNode` .	Formula parsing and evaluation
Batch Operation	A group of individual cell modifications treated as a single atomic action for undo/redo purposes. Implemented via the `CompoundCommand` type which contains multiple `Command` objects.	Undo/redo functionality
Cell	The fundamental data unit of the spreadsheet, containing value, formula, formatting metadata, and dependency information. Represented by the `Cell` type with fields: `value` , `formula` , `format` , and `dependencies` .	Core data model
CellFormat	A structured type defining visual presentation properties for a cell, including number format, decimal places, alignment, and font styles. Fields: `numberFormat` , `decimalPlaces` , `alignment` , `bold` , `italic` .	Cell formatting and display
Circular Reference	A dependency chain that forms a cycle (e.g., A1 references B1, B1 references C1, C1 references A1). Detected by the `detectCircularReference` method to prevent infinite recalculation loops.	Dependency graph and recalculation
ClipboardData	A structured representation of copied cells that preserves formulas, values, formatting, and original positions. Used by the `ClipboardManager` to support copy/paste operations with reference adjustment.	Copy/paste functionality
Command Pattern	A behavioral design pattern that encapsulates operations as objects, allowing for undo/redo functionality, logging, and queuing of operations. Implemented through the `Command` base class and specific command types like `SetCellValueCommand` .	Undo/redo system
Compound Command	A command that groups multiple atomic commands into a single undoable operation. The `CompoundCommand` type contains an array of individual `Command` objects, executed and undone as a unit.	Batch operations in undo/redo
CSV (Comma-Separated Values)	A simple file format for tabular data where values are separated by commas and rows by newlines. Our system provides `exportToCSV` and `importFromCSV` methods for data exchange with other applications.	Import/export functionality
Deep Copy	Creating a complete duplicate of an object, including all nested objects, as opposed to copying only the top-level reference. Essential for creating independent copies of cell data in the clipboard and history system.	State management, copy/paste

Term	Definition	Context
Dependency Graph	A directed graph tracking which cells depend on which other cells for their values. Implemented by the `DependencyGraph` type with `nodes` (cells) and `edges` (dependencies), enabling efficient recalculation.	Core spreadsheet engine
Dependency Injection	A design pattern where a component's dependencies are provided externally rather than created internally, promoting testability and loose coupling. Used throughout the architecture to connect components.	System architecture
Dirty Flag	A marker indicating that a cell needs recalculation because one of its dependencies has changed. While not explicitly stored as a field, this concept guides the recalculation algorithm which only processes affected cells.	Recalculation optimization
Displacement Vector	The row and column differences between source and target locations when copying cells. Used by the `adjustFormula` method to modify relative references during paste operations.	Copy/paste with reference adjustment
DOM Recycling	A performance optimization technique where DOM elements are reused for different logical content as the user scrolls, rather than creating and destroying elements. Core to the `VirtualGrid` implementation.	Grid rendering optimization
Edge Case	An unusual but valid scenario requiring special handling, such as empty cells in formulas, references to non-existent sheets, or formulas referencing themselves indirectly.	Error handling and validation
End-to-End Test	A test that verifies complete user workflows through the UI, simulating actual user interactions like typing formulas, copying cells, and undoing changes.	Testing strategy
Error Propagation	The process where errors in source cells (e.g., division by zero) cause errors in dependent cells, ensuring that invalid calculations don't produce misleading results.	Formula evaluation
Event Delegation	An event handling pattern where a single event listener is attached to a parent element (the grid container) to handle events for many child elements (cells), improving performance and memory efficiency.	Grid event handling
Format Preset	A predefined set of formatting properties with a name for easy reuse, such as "Currency", "Percentage", or "Header". Represented by the `FormatPreset` type containing a name and a `CellFormat` object.	Cell formatting system
Formula Parser	The component responsible for converting raw formula strings into evaluable expression trees. Implemented by the `FormulaParser` class with methods `tokenize`, `buildAST`, and `evaluateAST`.	Formula parsing and evaluation
Function Registry	A centralized map of function names to their implementations, allowing for extensible built-in functions. The `FunctionRegistry` type provides `register` and `get` methods for managing spreadsheet functions.	Formula evaluation
Integration Test	A test that verifies interactions between multiple components, such as ensuring that editing a cell triggers correct dependency updates and UI refreshes.	Testing strategy
Lexical Analysis	The process of converting a character stream (formula string) into tokens (numbers, operators, references, etc.). Performed by the `tokenize` or `scanFormula` method in the parser.	Formula parsing
Memoization	An optimization technique that caches previously computed results to avoid redundant work. Can be applied to formula parsing to avoid re-parsing unchanged	Performance optimization

Term	Definition	Context
	formulas.	
Mock	A test double that simulates the behavior of real objects, such as a mock value resolver for testing formula evaluation without a full spreadsheet. Created by the `createMockSpreadsheet` utility.	Testing utilities
Operator Precedence	Rules defining which operators are evaluated first in an expression (e.g., multiplication before addition). Implemented through the recursive descent parser's expression hierarchy.	Formula parsing
Operational Transformation	Algorithms for maintaining consistency in collaborative editors by transforming concurrent operations. Mentioned as a potential future extension for multi-user editing.	Future extensions
Panic Mode	An error recovery strategy in parsers where the parser discards tokens until it reaches a synchronization point (like a semicolon or line end) and continues parsing. Used in formula parsing for graceful error handling.	Error recovery in parsing
ParsedCellReference	A structured representation of a parsed cell address, including sheet name, row and column indices, and absolute reference flags. Fields: `sheetName`, `row`, `column`, `rowAbsolute`, `columnAbsolute`.	Cell address handling
ParsedFormula	The result of parsing a formula string, containing the AST, dependency list, original string, and any error information. Fields: `ast`, `dependencies`, `originalString`, `hasErrors`, `errorMessage`.	Formula parsing output
Reactive Dataflow	A system design where outputs automatically update when inputs change, without explicit update commands. The spreadsheet implements this through the dependency graph and automatic recalculation.	Core architecture principle
Recalculation Flow	The complete process from cell change to updated values in dependent cells, orchestrated by the `RecalculationFlow` component which coordinates parsing, graph updates, and topological sorting.	Core spreadsheet engine
Recursive Descent	A parsing technique that uses mutually recursive functions corresponding to grammar productions, where each function handles a specific syntactic construct. Used in our formula parser's `parseExpression` method.	Formula parsing
Reference Adjustment	The process of modifying cell references in formulas when they are copied to a new location. Relative references change based on displacement, while absolute references remain fixed. Performed by the `adjustFormula` method.	Copy/paste operations
Relative Reference	A cell reference that adjusts when copied to a new location (e.g., `A1` becomes `B2` when copied one row down and one column right). The default reference style in spreadsheets.	Copy/paste operations
Round-Trip Consistency	The property where exporting data (e.g., to CSV) and then re-importing it preserves all information, including formulas and formatting where supported by the format.	Import/export validation
Semantic Validation	Checking that a formula's meaning is valid, such as verifying that referenced cells exist, function arguments have appropriate types, and ranges are properly formed. Performed after syntactic validation.	Formula validation

Term	Definition	Context
Separation of Concerns	A design principle that separates a program into distinct sections with minimal overlap, such as dividing the system into presentation, business logic, and state management layers.	System architecture
Shallow Reference	Storing only a reference to an object rather than duplicating it, which is efficient but requires careful management to avoid unintended mutation. Contrasts with deep copy for state preservation.	Memory management
Sparse Array	A data structure that only stores elements that have values, saving memory by not allocating space for empty cells. The spreadsheet's grid uses nested Maps for sparse storage.	Data model optimization
Spreadsheet	The main application object representing a single worksheet, containing the cell grid, selection state, viewport position, and history. Represented by the `Spreadsheet` type.	Core application state
Syntax Validation	Checking that a formula's structure conforms to grammatical rules (proper operator placement, balanced parentheses, valid token sequences) before attempting semantic analysis.	Formula validation
Token	A basic lexical unit in formula parsing, such as a number, operator, cell reference, or function name. The `Token` type includes `type`, `value`, and `position` fields.	Formula parsing
Topological Sort	A linear ordering of nodes in a directed graph where dependencies come before dependents. Implemented by the `getTopologicalOrder` method to determine correct recalculation order.	Dependency graph algorithms
Type Coercion	Automatic conversion between data types during formula evaluation, such as converting string "42" to number 42 when used in arithmetic, with rules defined by the spreadsheet's evaluation engine.	Formula evaluation
Viewport	The visible portion of the larger virtual grid, defined by `topRow`, `leftCol`, `height`, and `width` properties. Only cells within the viewport are rendered to the DOM.	Virtual grid rendering
Virtual Scrolling	A rendering technique that displays only the visible elements while simulating the full scrollable content, dramatically improving performance for large grids. Core to the `VirtualGrid` implementation.	Grid rendering optimization
Workbook	A collection of worksheets, potentially with multiple sheets (tabs). Represented by the `Workbook` type containing a Map of sheet names to `Spreadsheet` objects and an active sheet name.	Multi-sheet extension
Worksheet	A single sheet within a workbook, synonymous with `Spreadsheet` in the basic implementation but distinguished in multi-sheet extensions. Also referred to as "Sheet".	Multi-sheet context

---

Convention Note: Throughout this document, these terms are used consistently according to these definitions. When implementing the spreadsheet system, maintaining consistency with these definitions will ensure clear communication and proper integration between components.

Implementation Guidance

Since this is a reference section, there is no implementation code to provide. However, the consistent use of these terms in code comments, documentation, and variable naming is recommended:

- Use TypeScript interfaces/types that match the glossary definitions
- Name variables and functions using the established terminology (e.g., `dependencyGraph` not `cellDeps`, `viewport` not `visibleArea`)
- Add JSDoc comments that reference these definitions when appropriate
- Consider creating a `types.ts` file that centralizes all type definitions using the exact names and structures from this glossary

---