Rich Text Editor with Collaboration Features

□ Table of Contents

* Scalability Considerations

Table of Contents

Clarify the Problem and Requirements

☐ Back to Top

Problem Understanding

☐ Back to Top

Design a sophisticated rich text editor that supports real-time collaboration, advanced formatting features, and AI-powered suggestions, similar to Notion, Microsoft Word Online, or Google Docs. The system must handle complex document structures, provide seamless collaboration experiences, and maintain document consistency across multiple concurrent users.

Functional Requirements

☐ Back to Top

- **Rich Text Formatting**: Bold, italic, underline, strikethrough, fonts, colors, sizes
- **Block-level Elements**: Headings, paragraphs, lists (ordered/unordered), blockquotes

- **Advanced Content**: Tables, images, videos, code blocks, mathematical equations
- **Real-time Collaboration**: Multi-user editing with conflict resolution and presence indicators
- **Document Structure**: Nested blocks, drag-and-drop reordering, collapsible sections
- **Comment System**: Threaded comments, suggestions, review workflow
- **AI Features**: Grammar checking, auto-completion, content suggestions, translation
- **Import/Export**: Support for various formats (Markdown, HTML, PDF, DOCX)

Non-Functional Requirements

☐ Back to Top

- **Performance**: <100ms keystroke response time, <50ms collaboration updates
- **Scalability**: Support 50+ concurrent editors per document, documents up to 100MB
- **Availability**: 99.9% uptime with offline editing capabilities
- **Consistency**: Eventual consistency across all clients with conflict resolution
- **Cross-platform**: Web browsers, mobile apps with feature parity
- **Accessibility**: WCAG 2.1 AA compliance, screen reader support
- **Security**: Content encryption, access controls, audit logging

Key Assumptions

☐ Back to Top

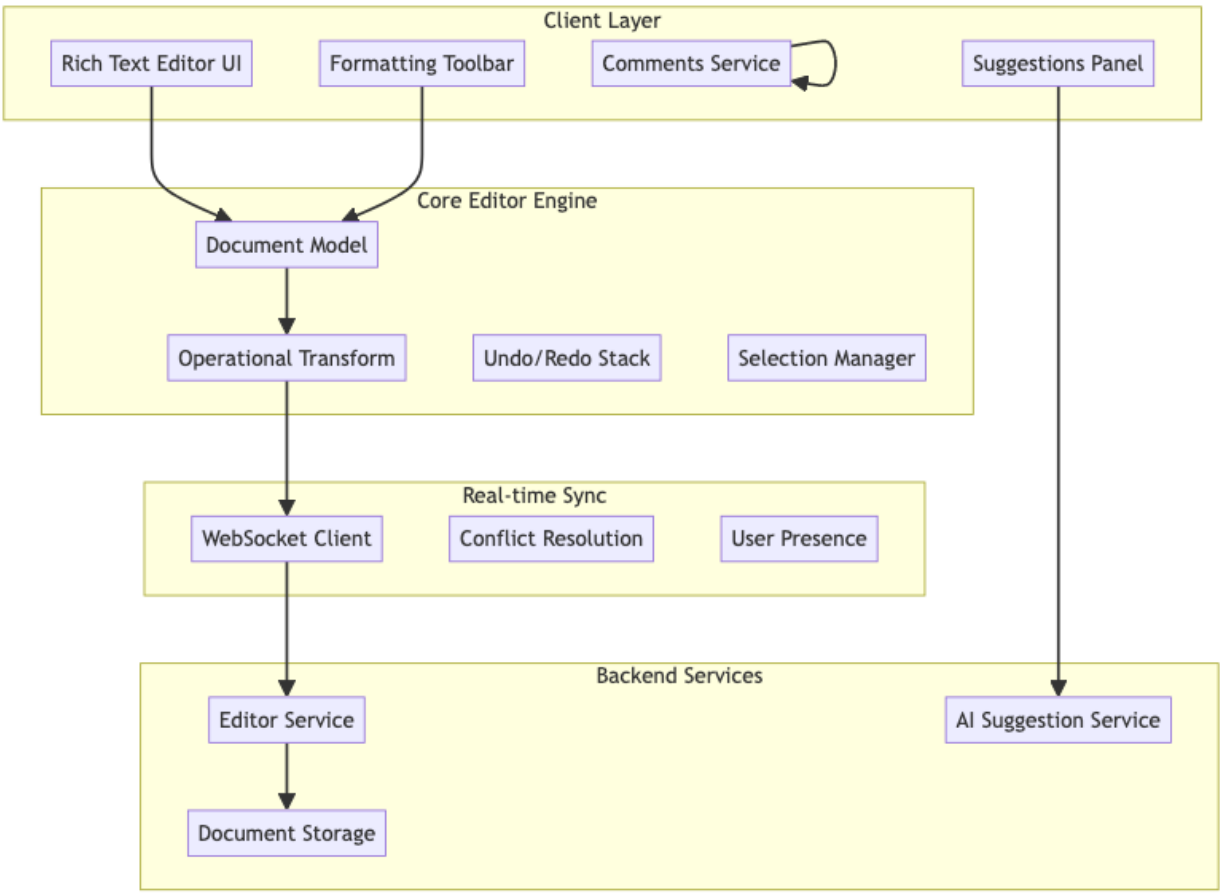
- Average document size: 1-10MB, maximum 100MB
- Typical editing session: 30-120 minutes
- Peak concurrent users per document: 20-50
- Operation frequency: 100-500 operations per minute during active editing
- Network conditions: Support for 3G to high-speed connections
- User base: Mixed technical proficiency levels

High-Level Design (HLD)

☐ Back to Top

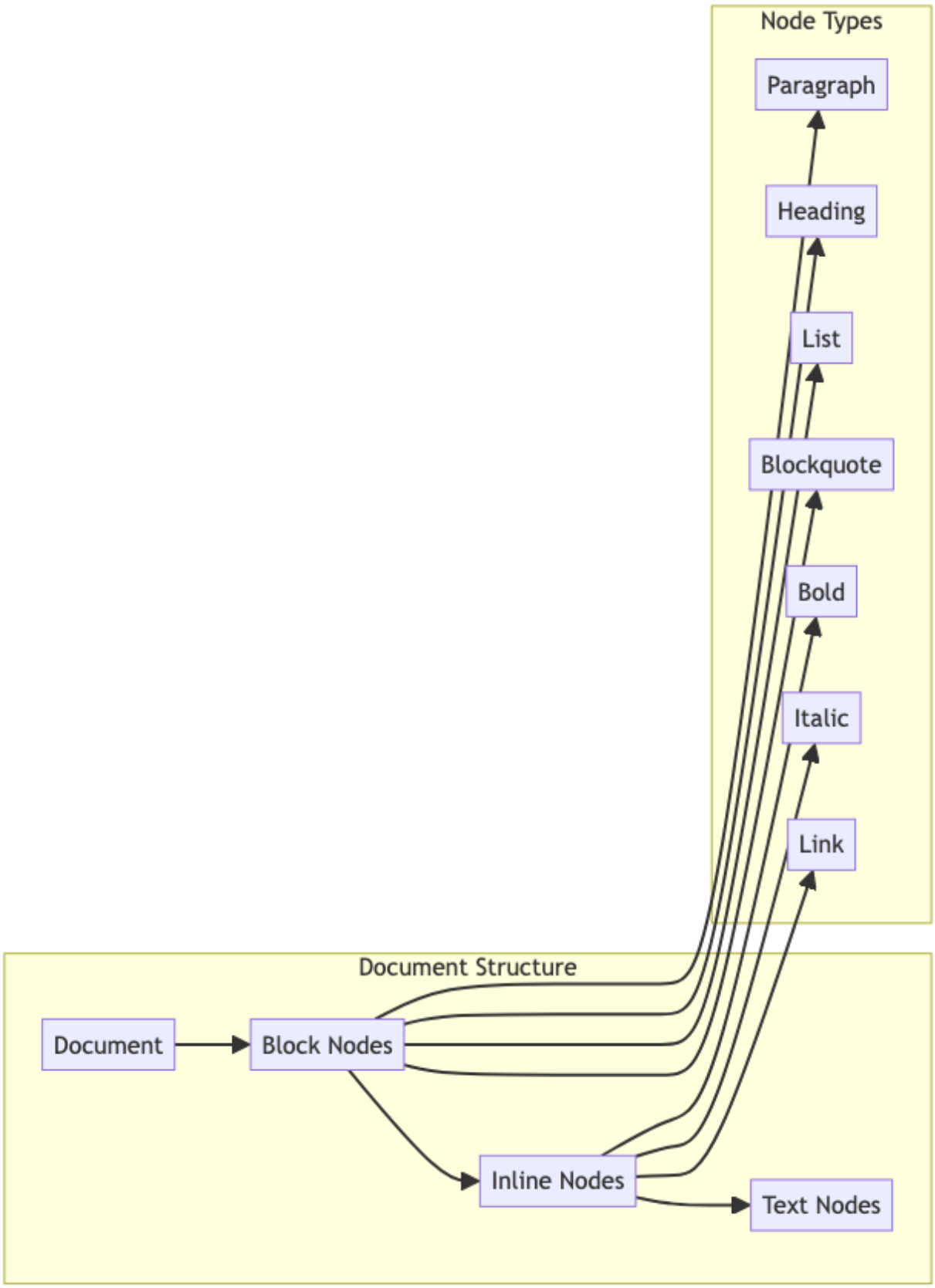
System Architecture Overview

☐ Back to Top



Document Model Architecture

Back to Top

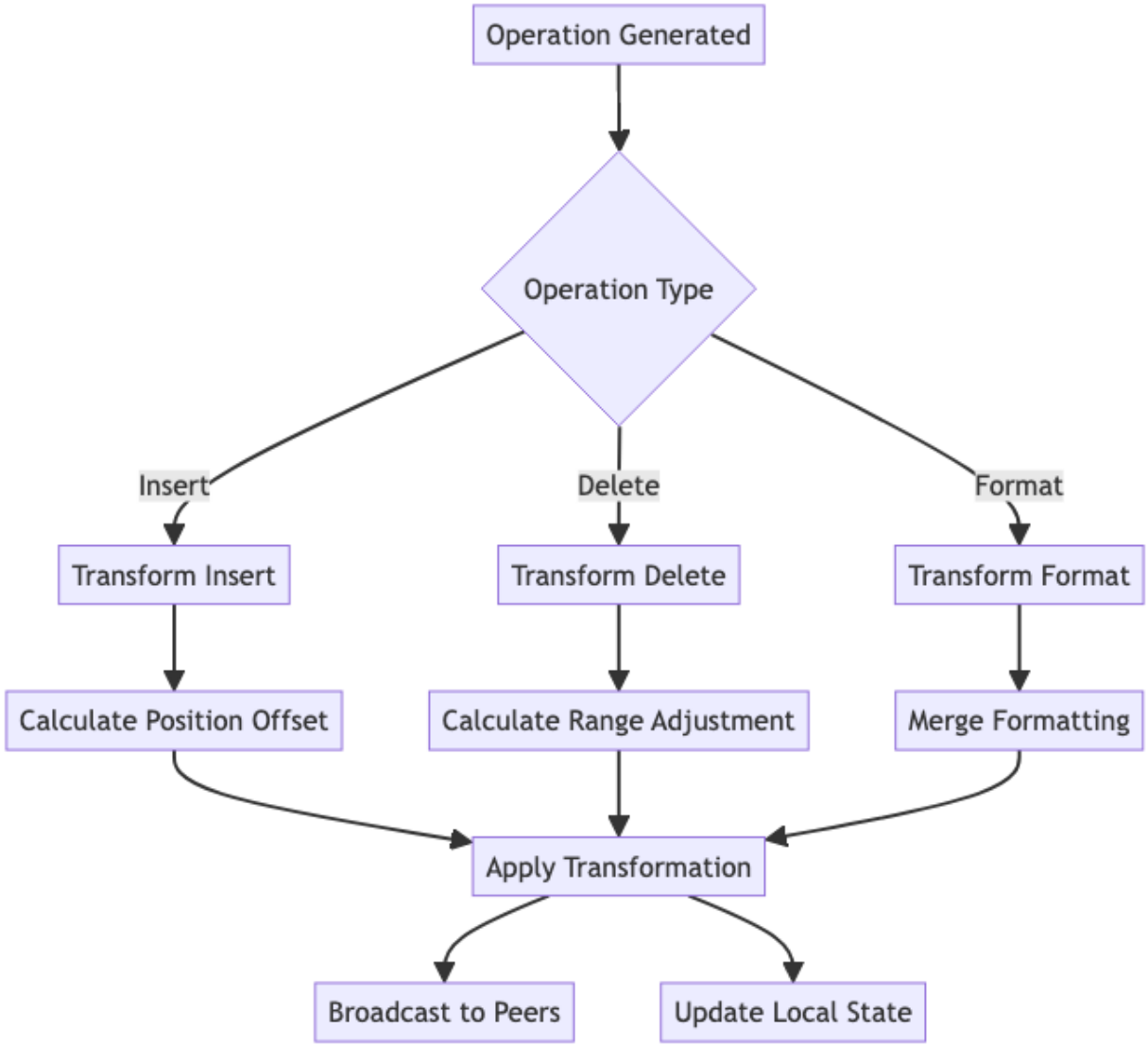

Node Types
Paragraph
Heading
List
Blockquote
Bold
Italic
Link
Document Structure
Document
Block Nodes
Inline Nodes
Text Nodes

Low-Level Design (LLD)

Back to Top

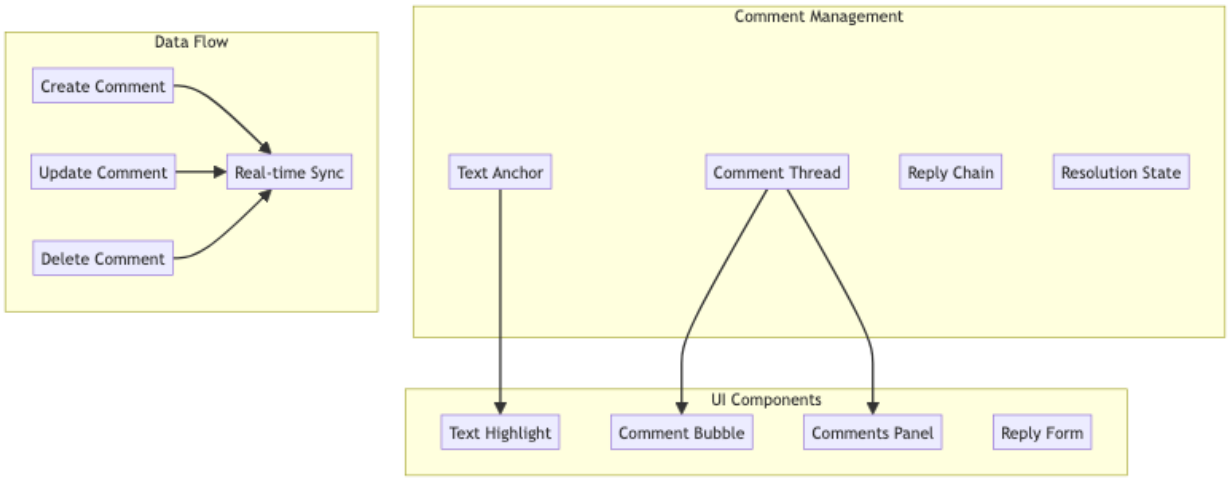
Operational Transform Algorithm

Back to Top



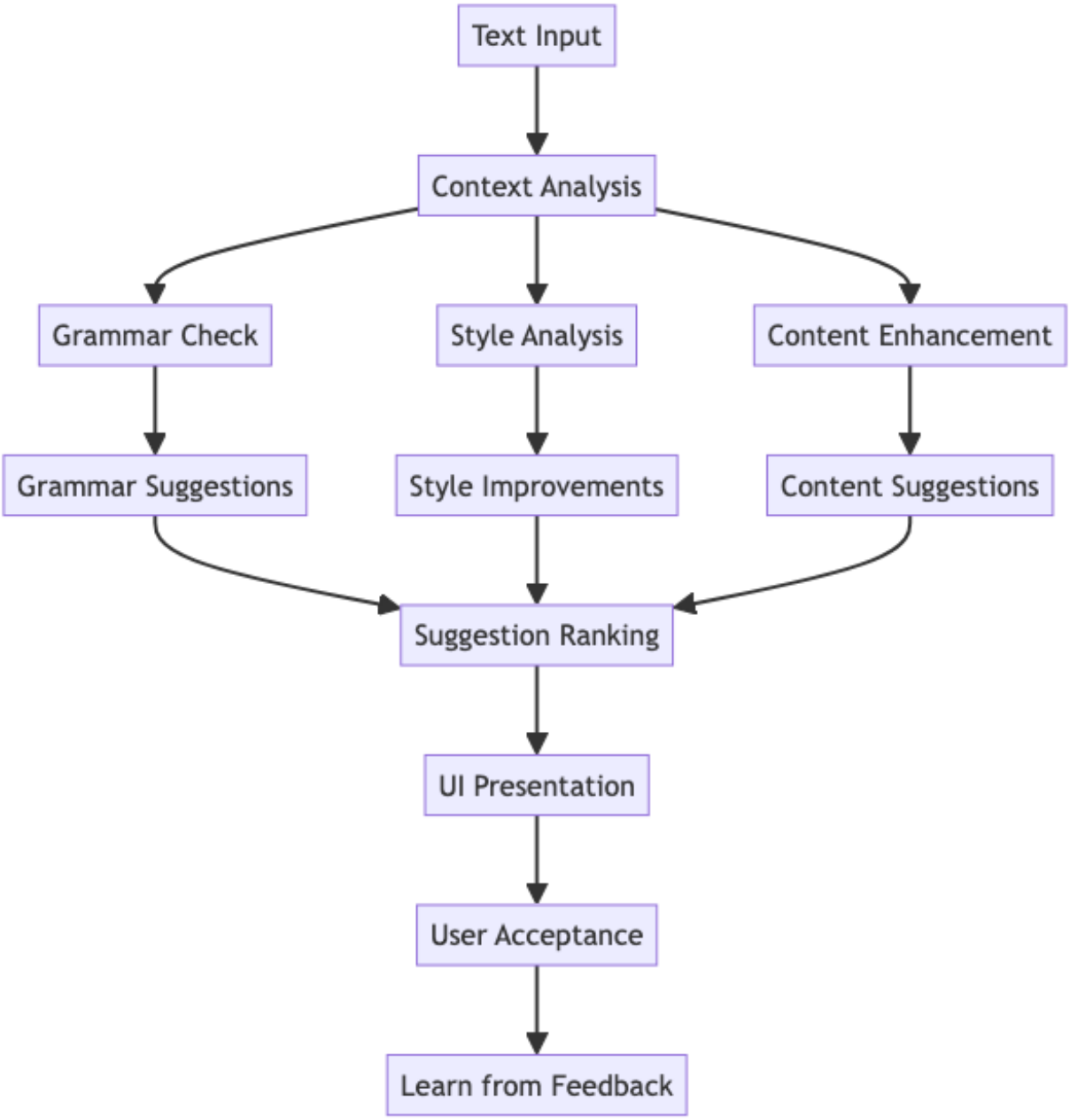
Comments System Architecture

Back to Top



AI Suggestions Engine

Back to Top



Core Algorithms

Back to Top

---

1. Operational Transform (OT) for Text Editing

Back to Top

Algorithm Purpose: Ensures consistency when multiple users edit simultaneously.

Key Components: - **Transform Function**: Adjusts operations based on concurrent changes - **State Vector**: Tracks document version for each client - **Operation Composition**: Combines multiple operations efficiently

Transform Logic:

```
For operations O1 and O2 occurring concurrently:
1. Calculate position offsets based on operation order
2. Adjust ranges for insertions/deletions
3. Merge formatting operations
4. Maintain intent preservation
```

Conflict Resolution Strategy: - Insert operations: Bias towards earlier timestamp - Delete operations: Check if range still exists - Format operations: Last-writer-wins with merge

2. Selection Synchronization Algorithm

Back to Top

Multi-user Selection Tracking:

```
Selection State = {
  userId: string,
  ranges: [{ start: position, end: position }],
  timestamp: number,
  cursor: position
}
```

Selection Transform Process: 1. Convert DOM selection to document model position 2. Apply operational transforms to maintain accuracy 3. Broadcast selection changes to peers 4. Render peer selections with user colors

3. Undo/Redo Stack Management

Back to Top

Command Pattern Implementation:

```
Command = {
  execute(): void,
  undo(): void,
```

```
  redo(): void,
  merge(other: Command): boolean
}
```

Stack Management Logic: - Group rapid operations (typing) into single commands - Maintain separate stacks for each user in collaborative mode - Implement command merging for efficiency - Handle conflicts with peer operations

4. Comment Anchoring Algorithm

Back to Top

Text Anchor Strategy:

```
Anchor = {
  startOffset: number,
  endOffset: number,
  contextBefore: string,
  contextAfter: string,
  nodeId: string
}
```

Anchor Maintenance Process: 1. Store relative positions within text nodes 2. Maintain context strings for fuzzy matching 3. Update anchors when text operations occur 4. Handle orphaned comments gracefully

5. AI Suggestion Ranking Algorithm

Back to Top

Ranking Factors: - Grammar importance score (0-1) - Style consistency impact (0-1) - User acceptance history (0-1) - Context relevance (0-1)

Ranking Formula:

```
Score = (0.4 × Grammar) + (0.3 × Style) + (0.2 × History) + (0.1 × Context)
```

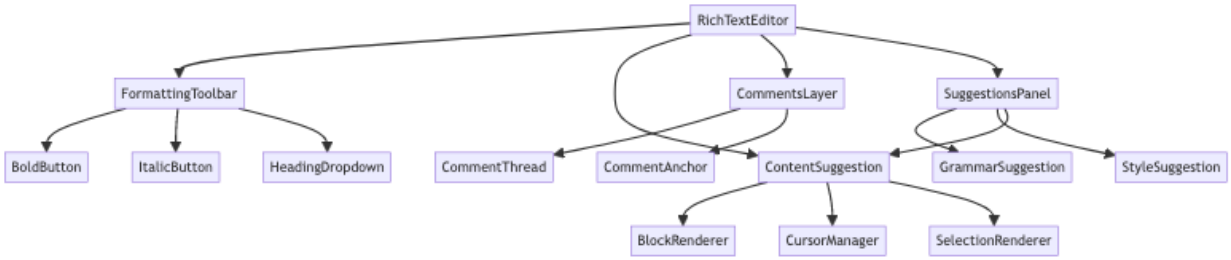
Learning Component: - Track user acceptance/rejection patterns - Adjust suggestion confidence scores - Personalize suggestions based on writing style

Component Architecture

Back to Top

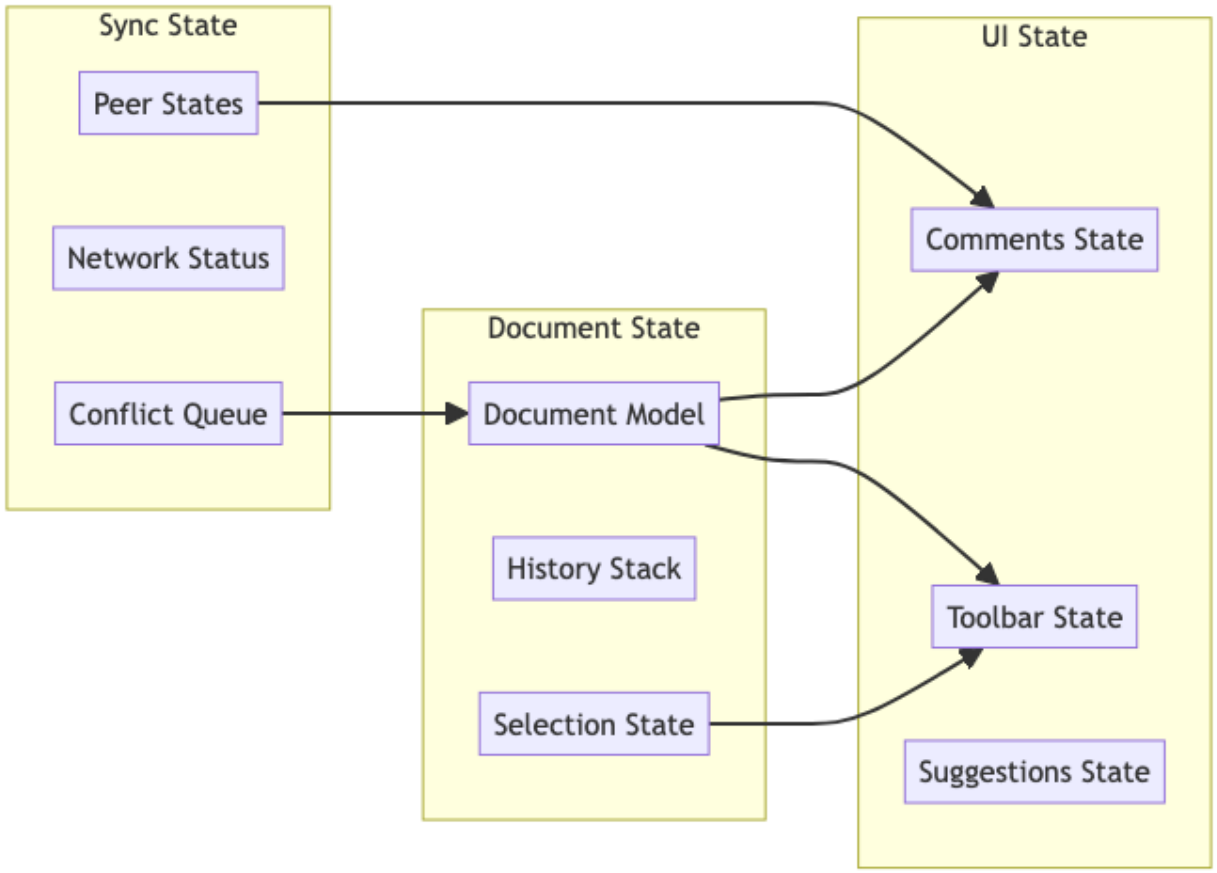
Editor Component Hierarchy

Back to Top



State Management Architecture

Back to Top



React Component Implementation

Back to Top

RichTextEditor.jsx

```jsx
import React, { useState, useCallback, useRef, useEffect } from 'react';
import { EditorProvider } from './EditorContext';
import FormattingToolbar from './FormattingToolbar';
import EditableContent from './EditableContent';
import { createEditorState, applyOperation } from './editorUtils';

const RichTextEditor = ({ initialContent = '', onSave, collaborative = false }) => {
  const [editorState, setEditorState] = useState(() => createEditorState(initialContent)
  const [selection, setSelection] = useState(null);
  const [history, setHistory] = useState({ undo: [], redo: [] });
  const [isComposing, setIsComposing] = useState(false);
  const editorRef = useRef(null);

  const handleEditorChange = useCallback((operation) => {
    setEditorState(prevState => {
      const newState = applyOperation(prevState, operation);

      // Add to history for undo/redo
      setHistory(prev => ({
        undo: [...prev.undo, prevState],
        redo: []
      }));

      return newState;
    });
  }, []);

  const handleFormat = useCallback((formatType, value) => {
    const operation = {
      type: 'format',
      formatType,
      value,
      selection
    };
    handleEditorChange(operation);
  }, [selection, handleEditorChange]);

  const handleUndo = useCallback(() => {
    if (history.undo.length > 0) {
      const previousState = history.undo[history.undo.length - 1];
      setHistory(prev => ({
        undo: prev.undo.slice(0, -1),
        redo: [editorState, ...prev.redo]
```

```
      }));
      setEditorState(previousState);
    }
  }, [history.undo, editorState]);

  return (
    <EditorProvider value={{
      editorState,
      selection,
      history,
      isComposing,
      onEditorChange: handleEditorChange,
      onFormat: handleFormat,
      onUndo: handleUndo
    }}>
      <div className="rich-text-editor" ref={editorRef}>
        <FormattingToolbar />
        <EditableContent />
      </div>
    </EditorProvider>
  );
};

export default RichTextEditor;
```

FormattingToolbar.jsx

```
import React, { useContext } from 'react';
import { EditorContext } from './EditorContext';
import ToolbarButton from './ToolbarButton';

const FormattingToolbar = () => {
  const { editorState, selection, onFormat, onUndo, history } = useContext(EditorContext

  const isFormatActive = (formatType) => {
    if (!selection) return false;
    return editorState.isFormatActive(formatType, selection);
  };

  return (
    <div className="formatting-toolbar">
      <ToolbarButton
        icon="B"
        title="Bold"
        isActive={isFormatActive('bold')}
        onClick={() => onFormat('bold')}
```

```
      />
      <ToolbarButton
        icon="I"
        title="Italic"
        isActive={isFormatActive('italic')}
        onClick={() => onFormat('italic')}
      />
      <ToolbarButton
        icon="U"
        title="Underline"
        isActive={isFormatActive('underline')}
        onClick={() => onFormat('underline')}
      />
    </div>
  );
};

export default FormattingToolbar;
```

EditableContent.jsx

```
import React, { useContext, useRef, useCallback } from 'react';
import { EditorContext } from './EditorContext';

const EditableContent = () => {
  const { editorState, onEditorChange } = useContext(EditorContext);
  const contentRef = useRef(null);

  const handleInput = useCallback((e) => {
    const operation = {
      type: 'input',
      data: e.data,
      inputType: e.inputType
    };
    onEditorChange(operation);
  }, [onEditorChange]);

  return (
    <div
      ref={contentRef}
      className="editable-content"
      contentEditable
      onInput={handleInput}
    >
      {editorState.document.blocks.map((block, index) => (
        <div key={block.id || index} className={`block-${block.type}`}>
```

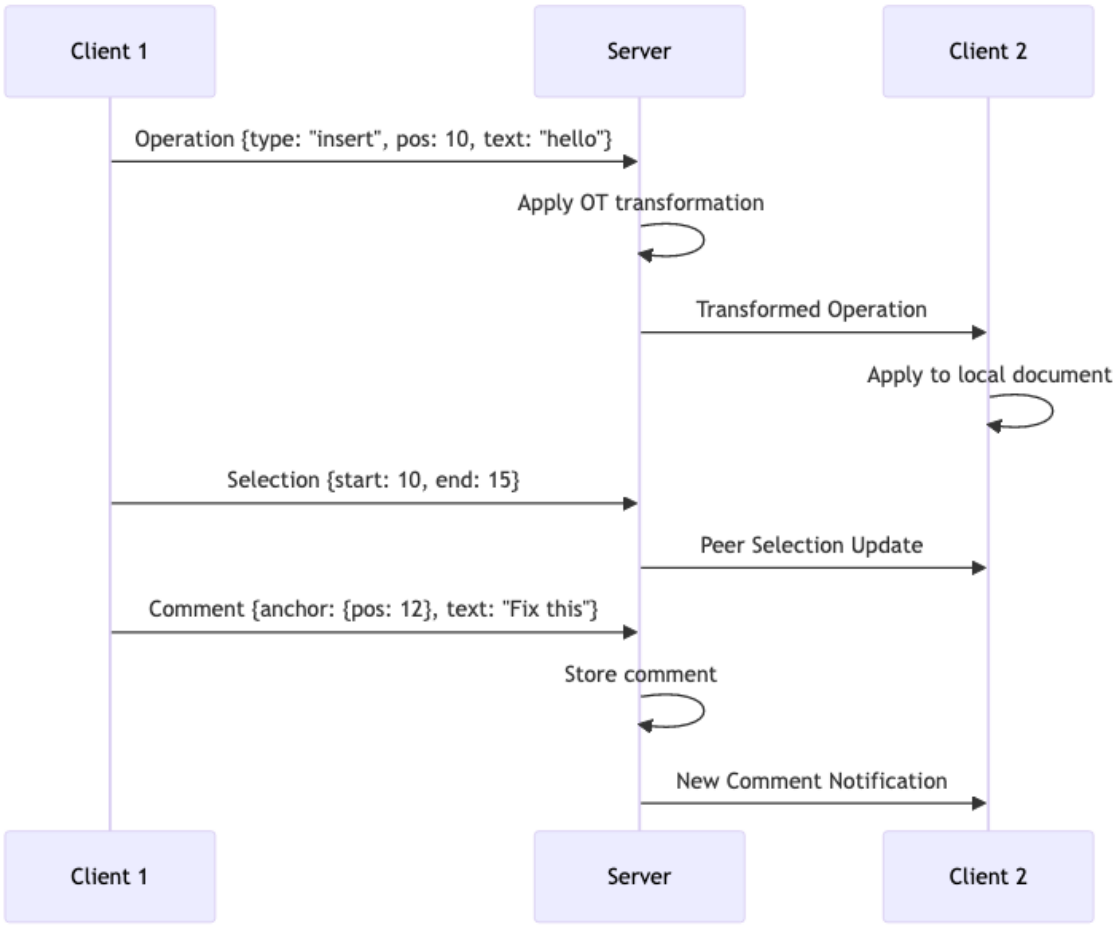
```
          {block.text}
        </div>
      ))}
    </div>
  );
};

export default EditableContent;
```

Real-time Synchronization

Back to Top

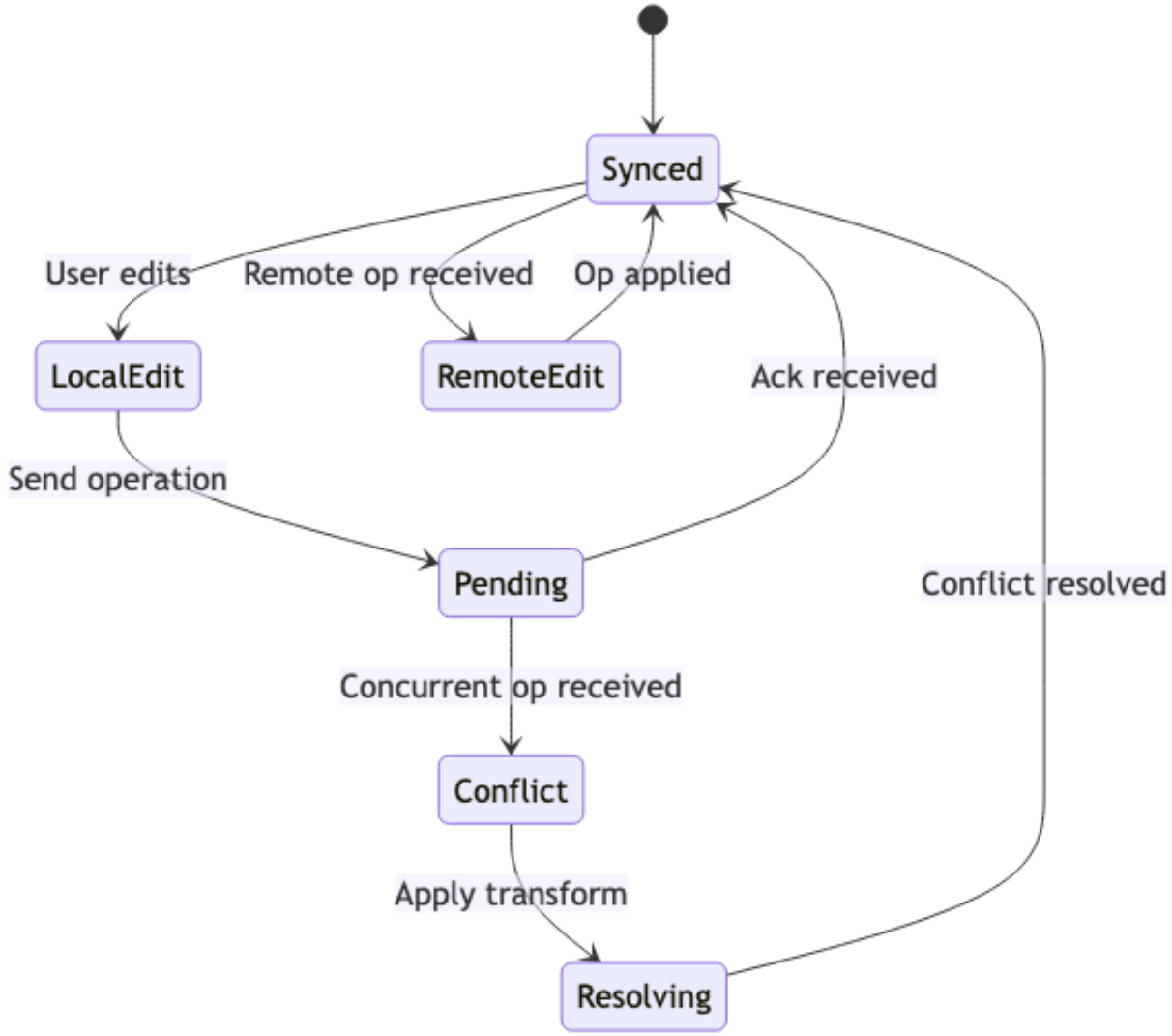
WebSocket Protocol Design

Back to Top



Conflict Resolution State Machine

Back to Top



TypeScript Interfaces & Component Props

Back to Top

Core Data Interfaces

```
interface EditorDocument {
  id: string;
  title: string;
```

```
  content: EditorState;
  collaborators: EditorUser[];
  version: number;
  lastModified: Date;
  permissions: DocumentPermissions;
  settings: DocumentSettings;
}

interface EditorOperation {
  id: string;
  type: 'insert' | 'delete' | 'format' | 'retain';
  position: number;
  content?: string;
  attributes?: TextAttributes;
  length?: number;
  authorId: string;
  timestamp: Date;
  clientId: string;
}

interface EditorUser {
  id: string;
  name: string;
  email: string;
  avatar?: string;
  color: string;
  cursor?: CursorState;
  selection?: SelectionRange;
  isActive: boolean;
}

interface TextAttributes {
  bold?: boolean;
  italic?: boolean;
  underline?: boolean;
  strikethrough?: boolean;
  fontSize?: number;
  fontFamily?: string;
  color?: string;
  backgroundColor?: string;
  link?: string;
}

interface Comment {
  id: string;
```

```
  documentId: string;
  position: number;
  length: number;
  content: string;
  authorId: string;
  timestamp: Date;
  replies: CommentReply[];
  resolved: boolean;
}
```

Component Props Interfaces

```
interface RichTextEditorProps {
  documentId: string;
  initialContent?: EditorState;
  readOnly?: boolean;
  placeholder?: string;
  theme?: 'light' | 'dark';
  onContentChange?: (content: EditorState) => void;
  onSelectionChange?: (selection: SelectionRange) => void;
  onError?: (error: EditorError) => void;
  autoSave?: boolean;
  spellCheck?: boolean;
}

interface EditorToolbarProps {
  editorState: EditorState;
  selection: SelectionRange;
  onFormatToggle: (format: string, value?: any) => void;
  onBlockTypeChange: (blockType: string) => void;
  onHistoryAction: (action: 'undo' | 'redo') => void;
  disabled?: boolean;
  customTools?: ToolbarItem[];
}

interface CollaborationBarProps {
  users: EditorUser[];
  currentUser: EditorUser;
  onInviteUser?: (email: string) => void;
  onUserClick?: (userId: string) => void;
  showPresence?: boolean;
  maxVisibleUsers?: number;
}

interface CommentSidebarProps {
```

```
  comments: Comment[];
  selectedCommentId?: string;
  onCommentAdd: (position: number, content: string) => void;
  onCommentReply: (commentId: string, content: string) => void;
  onCommentResolve: (commentId: string) => void;
  onCommentSelect: (commentId: string) => void;
}
```

API Reference

☐ Back to Top

Document Management

- `GET /api/documents` - List user's documents with collaboration status
- `POST /api/documents` - Create new document with initial content
- `GET /api/documents/:id` - Get document content and metadata
- `PUT /api/documents/:id/content` - Update document content with operations
- `DELETE /api/documents/:id` - Delete document and all associated data

Real-time Collaboration

- `WS /api/documents/:id/collaborate` - WebSocket for real-time editing
- `POST /api/documents/:id/operations` - Submit editing operation
- `GET /api/documents/:id/operations` - Get operation history with pagination
- `POST /api/documents/:id/transform` - Transform operations for conflict resolution
- `PUT /api/documents/:id/cursor` - Update user cursor and selection

Content Operations

- `POST /api/documents/:id/blocks` - Insert new content blocks (images, tables)
- `PUT /api/documents/:id/format` - Apply formatting to text selection
- `POST /api/documents/:id/search` - Search within document content
- `POST /api/documents/:id/replace` - Find and replace text with formatting
- `GET /api/documents/:id/export` - Export document to various formats

Comments & Reviews

- `POST /api/documents/:id/comments` - Add comment to specific document position
- `GET /api/documents/:id/comments` - Get all comments with thread support
- `PUT /api/comments/:id` - Update comment content or resolve status
- `DELETE /api/comments/:id` - Delete comment and all replies
- `POST /api/comments/:id/replies` - Reply to existing comment

Collaboration Features

- `POST /api/documents/:id/share` - Share document with permission levels
- `GET /api/documents/:id/collaborators` - Get document collaborators list
- `PUT /api/documents/:id/permissions` - Update user permissions for document
- `DELETE /api/documents/:id/collaborators/:userId` - Remove collaborator access
- `POST /api/documents/:id/suggestions` - Submit content suggestions for review

Version History

- `GET /api/documents/:id/versions` - Get document version history
- `GET /api/documents/:id/versions/:versionId` - Get specific version content
- `POST /api/documents/:id/restore` - Restore document to previous version
- `POST /api/documents/:id/compare` - Compare two document versions
- `GET /api/documents/:id/changes` - Get detailed change tracking

AI & Smart Features

- `POST /api/documents/:id/ai/complete` - AI-powered text completion
- `POST /api/documents/:id/ai/grammar` - Grammar and spell checking
- `POST /api/documents/:id/ai/summarize` - Generate content summary
- `POST /api/documents/:id/ai/translate` - Translate document content
- `POST /api/documents/:id/ai/suggest` - Get writing suggestions and improvements

Performance Optimizations

☐ Back to Top

Virtual Rendering for Large Documents

☐ Back to Top

Viewport-based Rendering: - Render only visible blocks plus buffer - Implement incremental DOM updates - Use document fragments for efficient insertion - Maintain block-level virtualization

Memory Management: - Lazy load historical operations - Compress old document states - Implement LRU cache for rendered blocks - Garbage collect unused command objects

Debouncing and Batching

☐ Back to Top

Operation Batching Strategy:

```
Batch Window = 50ms
Max Batch Size = 10 operations
Batch Types: [typing, formatting, selection]
```

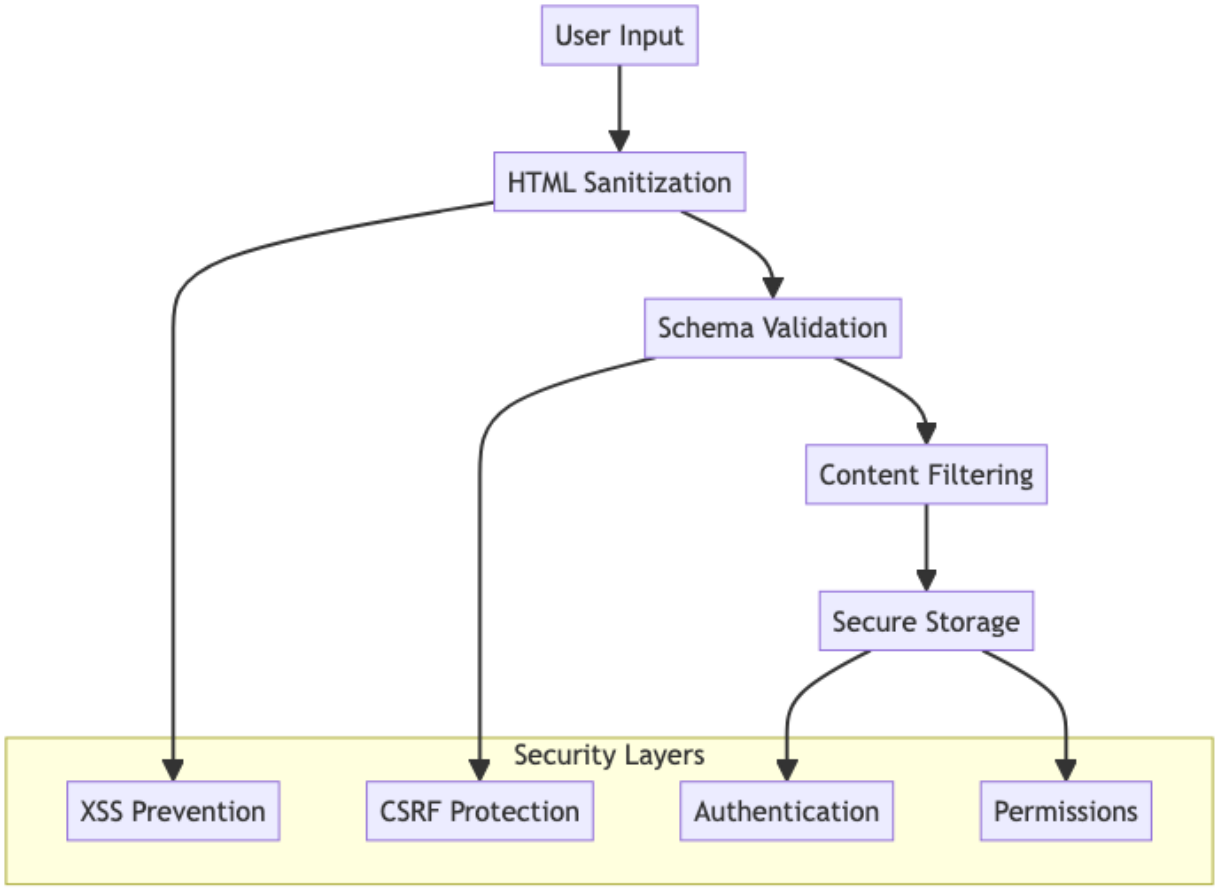
Network Optimization: - Compress operations using binary encoding - Implement delta compression for large changes - Use connection pooling for multiple documents - Implement smart reconnection with exponential backoff

Security Considerations

☐ Back to Top

Content Security Framework

☐ Back to Top



Permission Model

Back to Top

Document-level Permissions: - Owner: Full edit, share, delete rights - Editor: Edit content, add comments - Commenter: Read, add comments only - Viewer: Read-only access

Operation-level Security: - Validate user permissions before applying operations - Encrypt sensitive document content - Implement audit logging for all changes - Rate limiting for API operations

Testing Strategy

Back to Top

Unit Testing Focus Areas

Back to Top

Core Algorithm Testing: - Operational transform correctness - Selection synchronization accuracy - Comment anchoring stability - Undo/redo stack integrity

Component Testing: - Editor rendering performance - Toolbar state synchronization - Comments UI interactions - Suggestions acceptance flow

Integration Testing

Back to Top

Real-time Collaboration: - Multi-user editing scenarios - Network failure recovery - Conflict resolution accuracy - Performance under load

End-to-End Testing: - Complete editing workflows - Cross-browser compatibility - Mobile responsiveness - Accessibility compliance

Accessibility Implementation

Back to Top

Keyboard Navigation

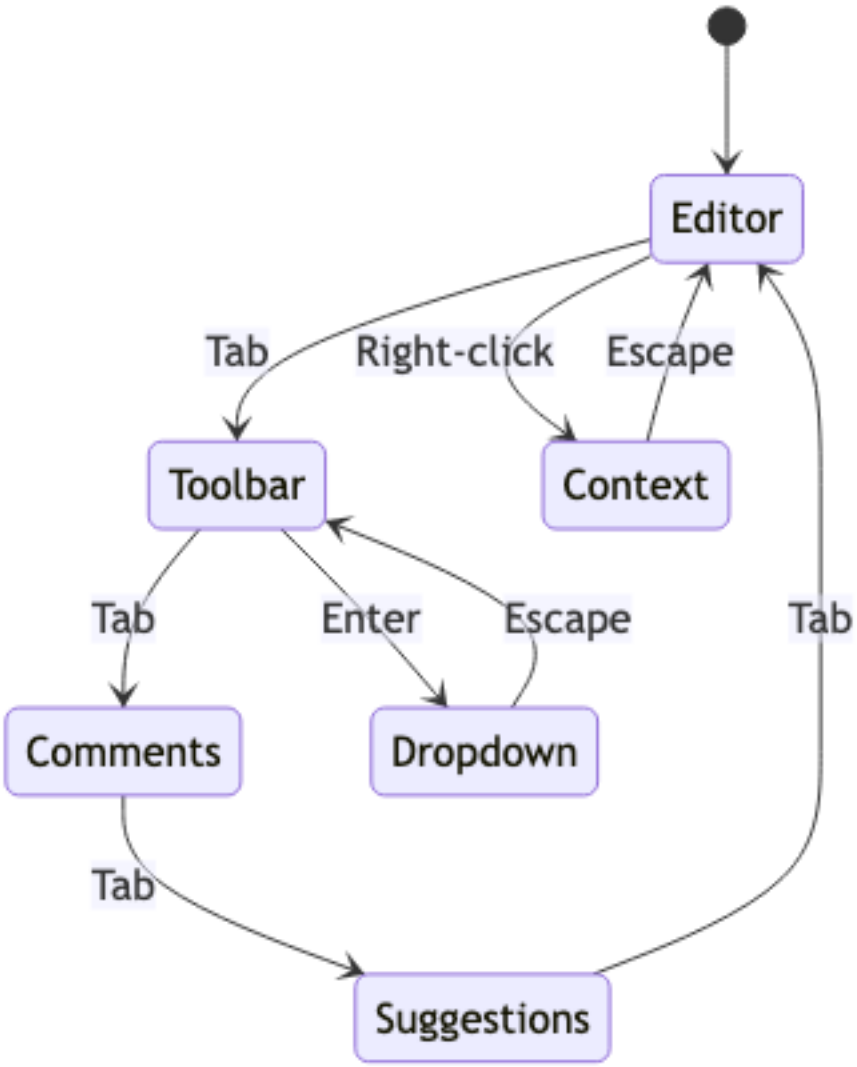
Back to Top

Navigation Patterns: - Arrow keys for cursor movement - Tab for toolbar navigation - Enter for line breaks - Shift+Tab for reverse navigation

Screen Reader Support: - ARIA labels for all interactive elements - Live regions for dynamic content updates - Proper heading structure - Alt text for embedded media

Focus Management

Back to Top



Trade-offs and Considerations

Back to Top

Performance vs Features

Back to Top

- **Rich formatting**: Complex DOM structure impacts performance
- **Real-time sync**: Network overhead vs user experience
- **AI suggestions**: Processing time vs suggestion quality
- **Large documents**: Memory usage vs responsiveness

Consistency vs Availability

Back to Top

- **Strong consistency**: Ensures data integrity but may impact availability
- **Eventual consistency**: Better performance but potential conflicts
- **Hybrid approach**: Critical operations strongly consistent, others eventual

Scalability Considerations

Back to Top

- **Document size limits**: Prevent memory exhaustion
- **Concurrent user limits**: Maintain performance standards
- **Operation rate limiting**: Prevent abuse and ensure stability
- **Storage optimization**: Balance between features and cost

This rich text editor system provides a comprehensive foundation for collaborative document editing with advanced features like real-time synchronization, AI-powered suggestions, and robust comment systems while maintaining performance and accessibility standards.