Design a Real-Time Collaborative Text Editor (like Google Docs)

□ Table of Contents

Table of Contents

Clarify the Problem and Requirements

☐ Back to Top

Problem Understanding

☐ Back to Top

Design a real-time collaborative text editor enabling multiple users to simultaneously edit documents with conflict resolution, similar to Google Docs. Changes must be synchronized across all clients instantly with consistent document state.

Functional Requirements

Back to Top

- **Multi-user Real-time Editing**: 100+ concurrent users per document
- **Rich Text Formatting**: Bold, italic, headings, lists, links, images
- **Document Management**: Create, save, share, version history
- **User Presence**: Show active users and cursor positions
- **Comments & Suggestions**: Collaborative review features
- **Offline Support**: Local editing with sync on reconnection

Non-Functional Requirements

Back to Top

- **Latency**: <200ms for operation propagation
- **Consistency**: Eventual consistency across all clients
- **Availability**: 99.9% uptime with graceful degradation
- **Scalability**: Support millions of documents, thousands of concurrent users
- **Performance**: <2s document load time, instant local operations

Key Assumptions

Back to Top

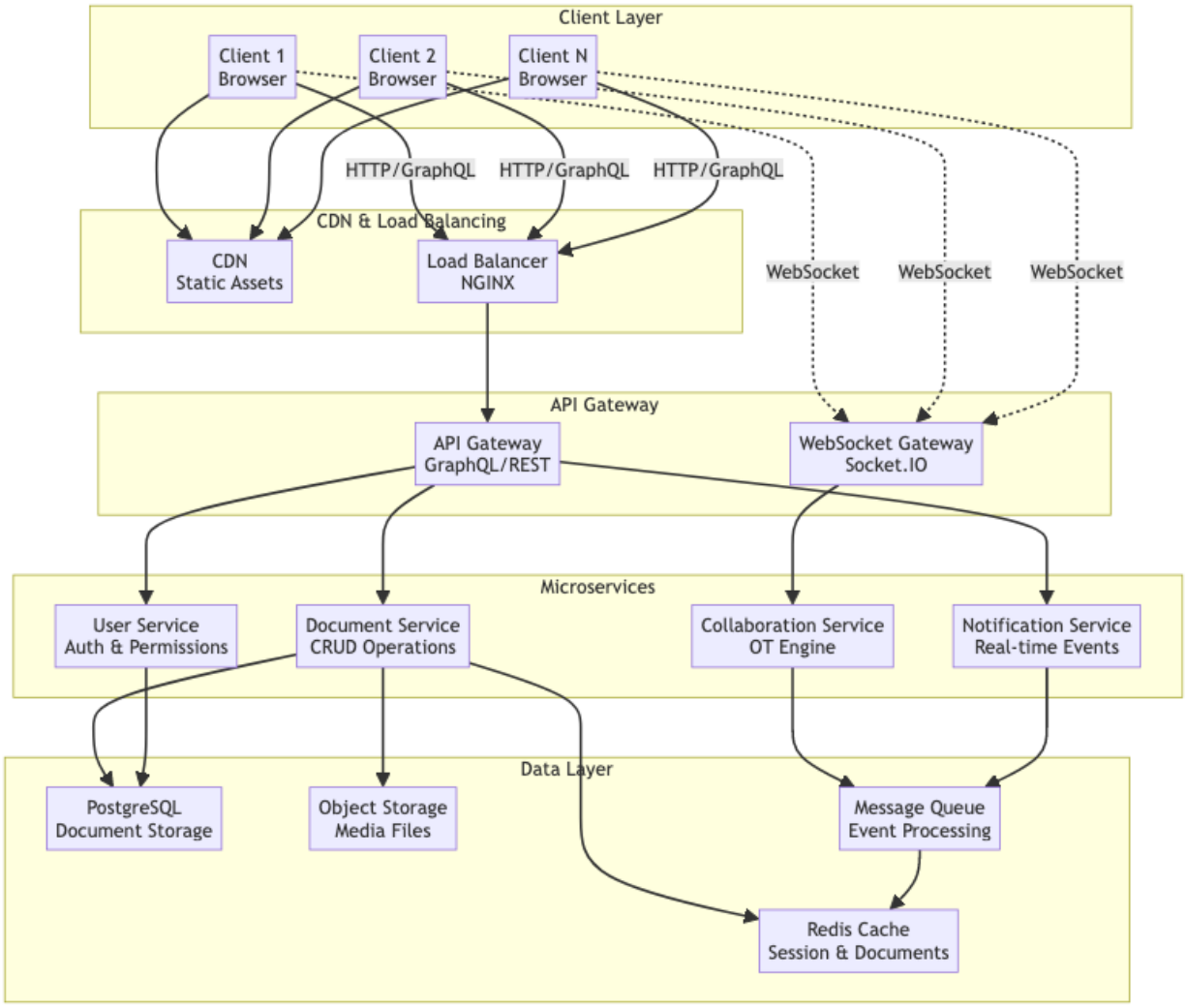
- Average document: 50KB, max 10MB
- Peak concurrent users per document: 100
- Operation frequency: 1000 ops/second for popular documents
- Network conditions: Handle 3G to fiber connections
- Browser support: Modern browsers with WebSocket support

High-Level Architecture

Back to Top

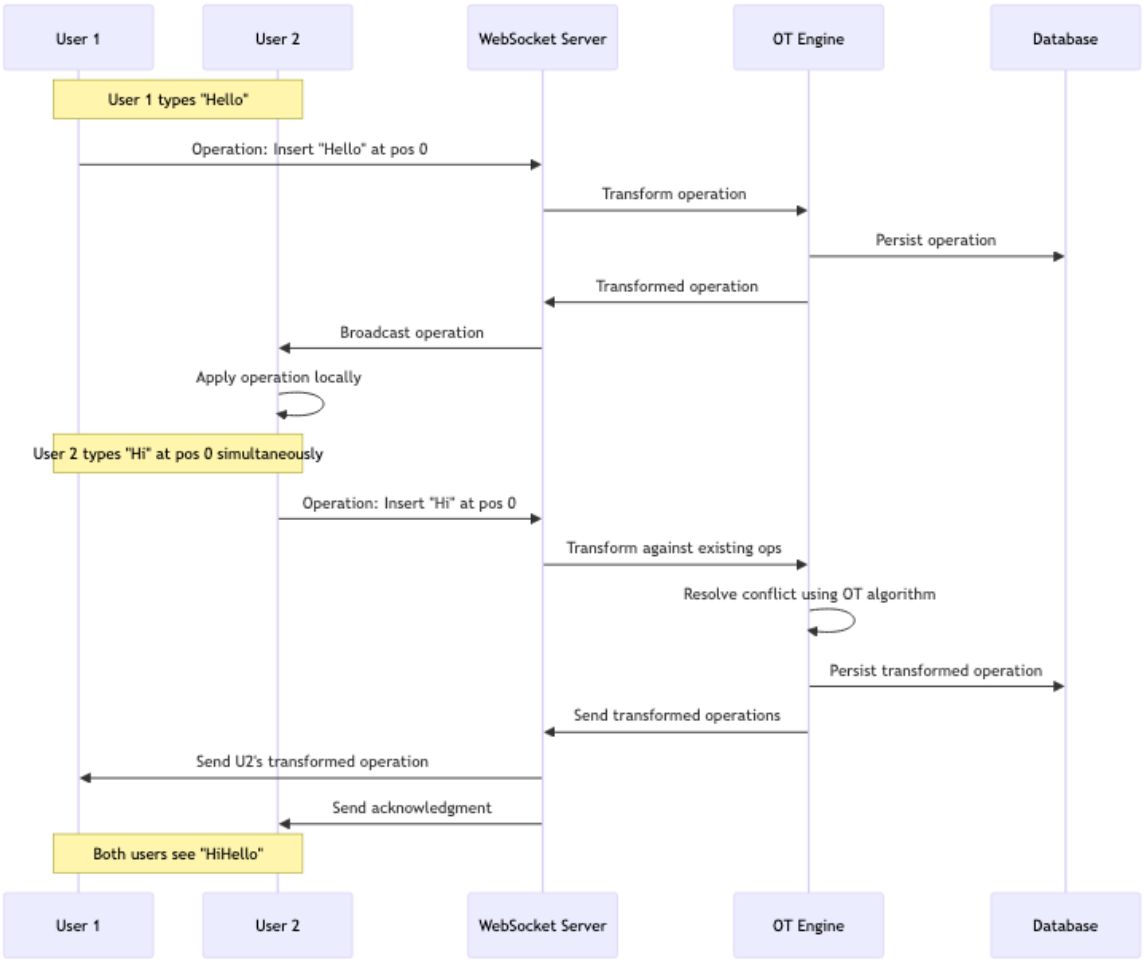
System Architecture Diagram

Back to Top



Data Flow Architecture

Back to Top

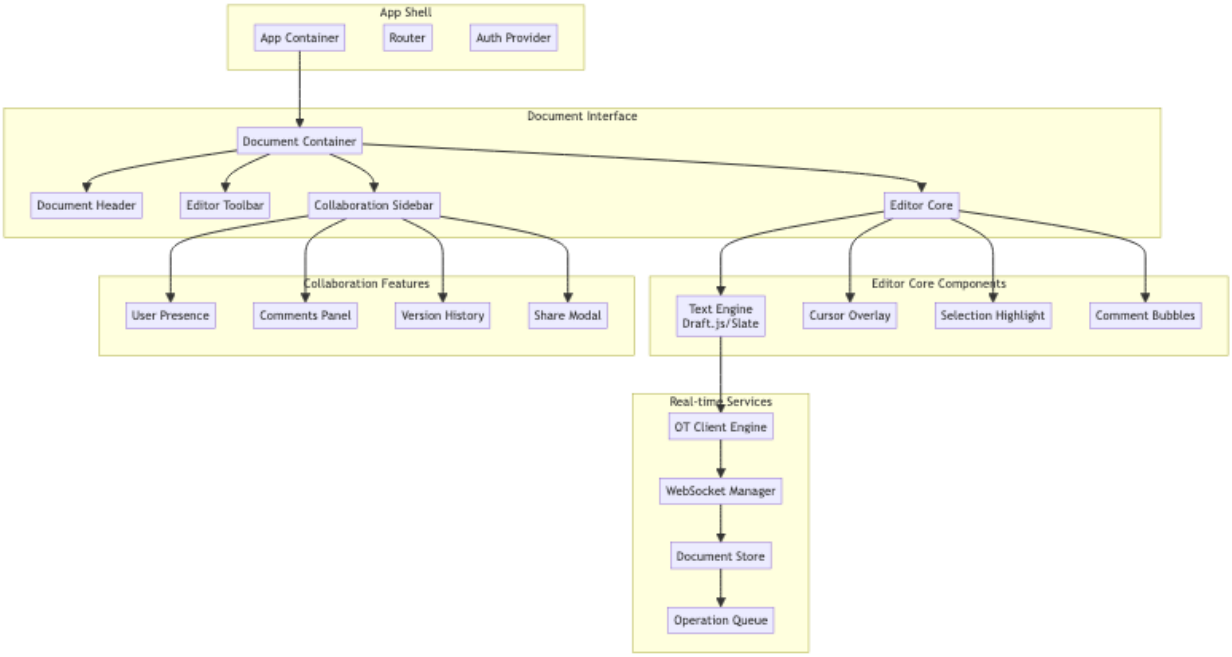


UI/UX and Component Structure

☐ Back to Top

Frontend Component Architecture

☐ Back to Top



React Component Implementation ☐ Back to Top

DocumentContainer.jsx

```
import React, { useState, useEffect, useRef } from 'react';
import { Editor, EditorState, ContentState } from 'draft-js';
import { CollaborationProvider } from './CollaborationContext';
import DocumentHeader from './DocumentHeader';
import EditorToolbar from './EditorToolbar';
import CollaborationSidebar from './CollaborationSidebar';
import CursorOverlay from './CursorOverlay';

const DocumentContainer = ({ documentId }) => {
  const [editorState, setEditorState] = useState(EditorState.createEmpty());
  const [activeUsers, setActiveUsers] = useState([]);
  const [operations, setOperations] = useState([]);
  const [isConnected, setIsConnected] = useState(false);
  const editorRef = useRef(null);
  const wsRef = useRef(null);

  useEffect(() => {
    // Initialize WebSocket connection
    wsRef.current = new WebSocket(`ws://localhost:8080/documents/${documentId}`);

    wsRef.current.onopen = () => {
```

```
    setIsConnected(true);
  };

  wsRef.current.onmessage = (event) => {
    const data = JSON.parse(event.data);
    handleRemoteOperation(data);
  };

  return () => {
    wsRef.current?.close();
  };
}, [documentId]);

const handleRemoteOperation = (operation) => {
  // Apply OT transformation and update editor state
  const newContentState = applyOperation(
    editorState.getCurrentContent(),
    operation
  );

  setEditorState(
    EditorState.push(editorState, newContentState, 'insert-characters')
  );
};

const handleLocalChange = (newEditorState) => {
  const currentContent = editorState.getCurrentContent();
  const newContent = newEditorState.getCurrentContent();

  if (currentContent !== newContent) {
    const operation = generateOperation(currentContent, newContent);
    sendOperation(operation);
  }

  setEditorState(newEditorState);
};

const sendOperation = (operation) => {
  if (wsRef.current && isConnected) {
    wsRef.current.send(JSON.stringify(operation));
  }
};

return (
  <CollaborationProvider value={{
```

```
      activeUsers,
      operations,
      isConnected,
      sendOperation
    }}>
      <div className="document-container">
        <DocumentHeader documentId={documentId} />
        <EditorToolbar
          editorState={editorState}
          onStateChange={setEditorState}
        />

        <div className="editor-workspace">
          <div className="editor-area">
            <CursorOverlay activeUsers={activeUsers} />
            <Editor
              ref={editorRef}
              editorState={editorState}
              onChange={handleLocalChange}
              placeholder="Start typing..."
            />
          </div>

          <CollaborationSidebar
            activeUsers={activeUsers}
            onUserInvite={(email) => console.log('Invite:', email)}
          />
        </div>
      </div>
    </CollaborationProvider>
  );
};

export default DocumentContainer;
```

EditorToolbar.jsx

```
import React from 'react';
import { RichUtils } from 'draft-js';

const EditorToolbar = ({ editorState, onStateChange }) => {
  const handleStyleToggle = (style) => {
    onStateChange(RichUtils.toggleInlineStyle(editorState, style));
  };

  const handleBlockToggle = (blockType) => {
```

```
  onStateChange(RichUtils.toggleBlockType(editorState, blockType));
};

const currentStyle = editorState.getCurrentInlineStyle();
const selection = editorState.getSelection();
const blockType = editorState
  .getCurrentContent()
  .getBlockForKey(selection.getStartKey())
  .getType();

return (
  <div className="editor-toolbar">
    <div className="toolbar-group">
      <button
        className={currentStyle.has('BOLD') ? 'active' : ''}
        onClick={() => handleStyleToggle('BOLD')}
      >
        Bold
      </button>
      <button
        className={currentStyle.has('ITALIC') ? 'active' : ''}
        onClick={() => handleStyleToggle('ITALIC')}
      >
        Italic
      </button>
      <button
        className={currentStyle.has('UNDERLINE') ? 'active' : ''}
        onClick={() => handleStyleToggle('UNDERLINE')}
      >
        Underline
      </button>
    </div>

    <div className="toolbar-group">
      <button
        className={blockType === 'header-one' ? 'active' : ''}
        onClick={() => handleBlockToggle('header-one')}
      >
        H1
      </button>
      <button
        className={blockType === 'header-two' ? 'active' : ''}
        onClick={() => handleBlockToggle('header-two')}
      >
        H2
```

```
        </button>
        <button
          className={blockType === 'unordered-list-item' ? 'active' : ''}
          onClick={() => handleBlockToggle('unordered-list-item')}
        >
          • List
        </button>
      </div>
    </div>
  );
};

export default EditorToolbar;
```

CursorOverlay.jsx

```
import React from 'react';

const CursorOverlay = ({ activeUsers }) => {
  return (
    <div className="cursor-overlay">
      {activeUsers.map(user => (
        <div
          key={user.id}
          className="remote-cursor"
          style={{
            left: user.cursorPosition?.x || 0,
            top: user.cursorPosition?.y || 0,
            borderColor: user.color
          }}
        >
          <div
            className="cursor-flag"
            style={{ backgroundColor: user.color }}
          >
            {user.name}
          </div>
        </div>
      ))}
    </div>
  );
};

export default CursorOverlay;
```

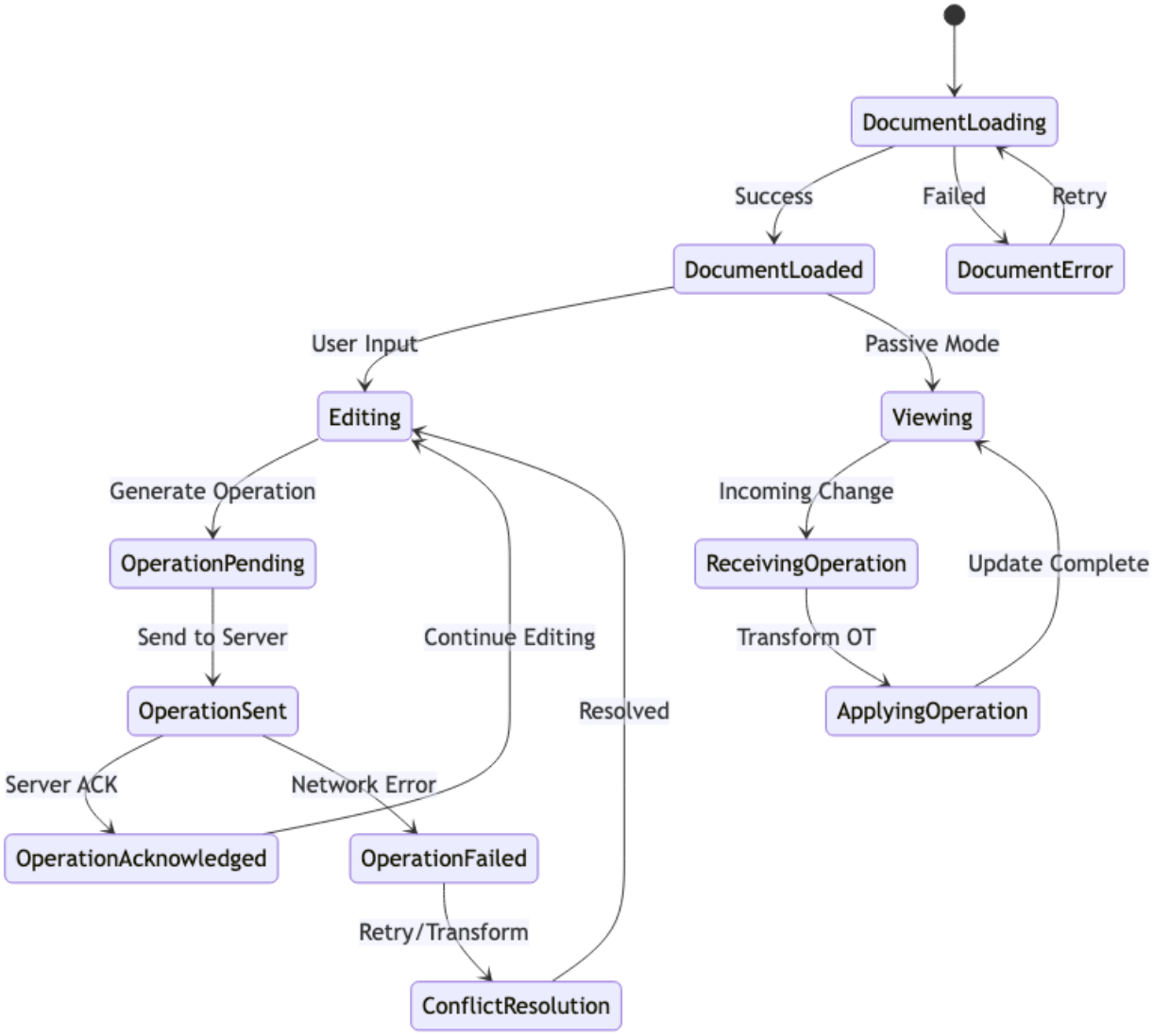
OT Engine Utilities

```javascript
// otEngine.js
export const generateOperation = (oldContent, newContent) => {
  // Simplified operation generation
  const oldText = oldContent.getPlainText();
  const newText = newContent.getPlainText();

  if (newText.length > oldText.length) {
    // Insert operation
    const insertIndex = findInsertIndex(oldText, newText);
    const insertedText = newText.slice(insertIndex, insertIndex + (newText.length - oldT

    return {
      type: 'insert',
      position: insertIndex,
      content: insertedText,
      timestamp: Date.now(),
      clientId: getClientId()
    };
  } else if (newText.length < oldText.length) {
    // Delete operation
    const deleteIndex = findDeleteIndex(oldText, newText);
    const deleteLength = oldText.length - newText.length;

    return {
      type: 'delete',
      position: deleteIndex,
      length: deleteLength,
      timestamp: Date.now(),
      clientId: getClientId()
    };
  }

  return null;
};

export const applyOperation = (content, operation) => {
  const text = content.getPlainText();

  switch (operation.type) {
    case 'insert':
      const newText = text.slice(0, operation.position) +
                      operation.content +
                      text.slice(operation.position);
      return ContentState.createFromText(newText);
```

```
    case 'delete':
      const deletedText = text.slice(0, operation.position) +
                          text.slice(operation.position + operation.length);
      return ContentState.createFromText(deletedText);

    default:
      return content;
  }
};

// Transform operation based on concurrent operations
export const transformOperation = (op1, op2) => {
  if (op1.type === 'insert' && op2.type === 'insert') {
    if (op1.position <= op2.position) {
      return { ...op2, position: op2.position + op1.content.length };
    }
    return op2;
  }

  if (op1.type === 'delete' && op2.type === 'insert') {
    if (op1.position < op2.position) {
      return { ...op2, position: op2.position - op1.length };
    }
    return op2;
  }

  // Add more transformation rules...
  return op2;
};
```

State Management Flow

☐ Back to Top



Real-Time Sync, Data Modeling & APIs

☐ Back to Top

Operational Transform Algorithm

☐ Back to Top

Core Concept: Operational Transform (OT) is a concurrency control algorithm that ensures consistency across multiple users editing the same document simultaneously. It

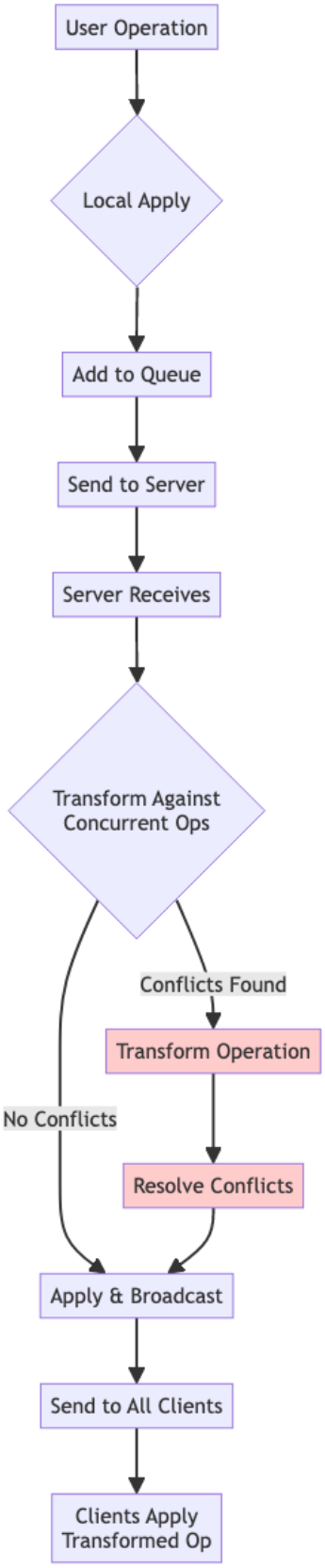
transforms operations based on concurrent changes, so that all clients eventually arrive at the same document state, even if operations arrive out of order. This approach typically requires a centralized server to manage and transform operations.

Example: OT in Action Consider two users, Alice and Bob, editing a document that initially contains “Hello”.

1. **Alice (Client A)** types ” World” at the end of “Hello” (index 5).
    - Alice’s local document: “Hello World”
    - Alice sends `OpA: Insert " World" at index 5` to the server.
2. **Bob (Client B)**, concurrently, types ” Awesome” at the beginning of “Hello” (index 0).
    - Bob’s local document: ” AwesomeHello”
    - Bob sends `OpB: Insert " Awesome" at index 0` to the server.
3. **Server receives OpA first.** It applies `OpA` to its document state: “Hello World”.
4. **Server then receives OpB.** Before applying `OpB`, the server notices that `OpB` was generated based on an older state of the document (before ” World” was added). The server uses OT to transform `OpB` against `OpA`.
    - `OpB` (Insert ” Awesome” at index 0) needs to be adjusted because ” World” was inserted at index 5.
    - The transformed `OpB'` becomes `Insert " Awesome" at index 0`. (In this specific case, the position remains the same as it’s an insertion at the beginning).
5. **Server applies transformed OpB’** to its document: ” AwesomeHello World”.
6. **Server sends transformed operations back to clients.**
    - To Alice, the server sends Bob’s transformed operation (`OpB': Insert " Awesome" at index 0`). Alice applies this to her document: ” AwesomeHello World”.
    - To Bob, the server sends Alice’s original operation (`OpA: Insert " World" at index 5`). Bob applies this, but critically, he needs to transform `OpA` against his own `OpB` that was applied locally. The transformed `OpA'` becomes `Insert " World" at index 13` (original index 5 + 8 characters from ” Awesome”). Bob applies this to his document: ” AwesomeHello World”.

Result: Both Alice and Bob consistently see ” AwesomeHello World”. The OT algorithm ensured that concurrent operations were correctly integrated without manual conflict resolution by the users.

OT Algorithm Flow □ Back to Top


User Operation
Local Apply
Add to Queue
Send to Server
Server Receives
Transform Against Concurrent Ops
Conflicts Found
Transform Operation
No Conflicts
Resolve Conflicts
Apply & Broadcast
Send to All Clients
Clients Apply Transformed Op

Key OT Transformation Rules ☐ Back to Top

---

1. **Insert vs Insert**:
   - Same position: Use timestamp/user priority
   - Different positions: Adjust positions based on order
2. **Insert vs Delete**:
   - Delete before insert: Adjust insert position
   - Insert before delete: Adjust delete position and length
3. **Delete vs Delete**:
   - Overlapping: Merge delete ranges
   - Non-overlapping: Adjust positions

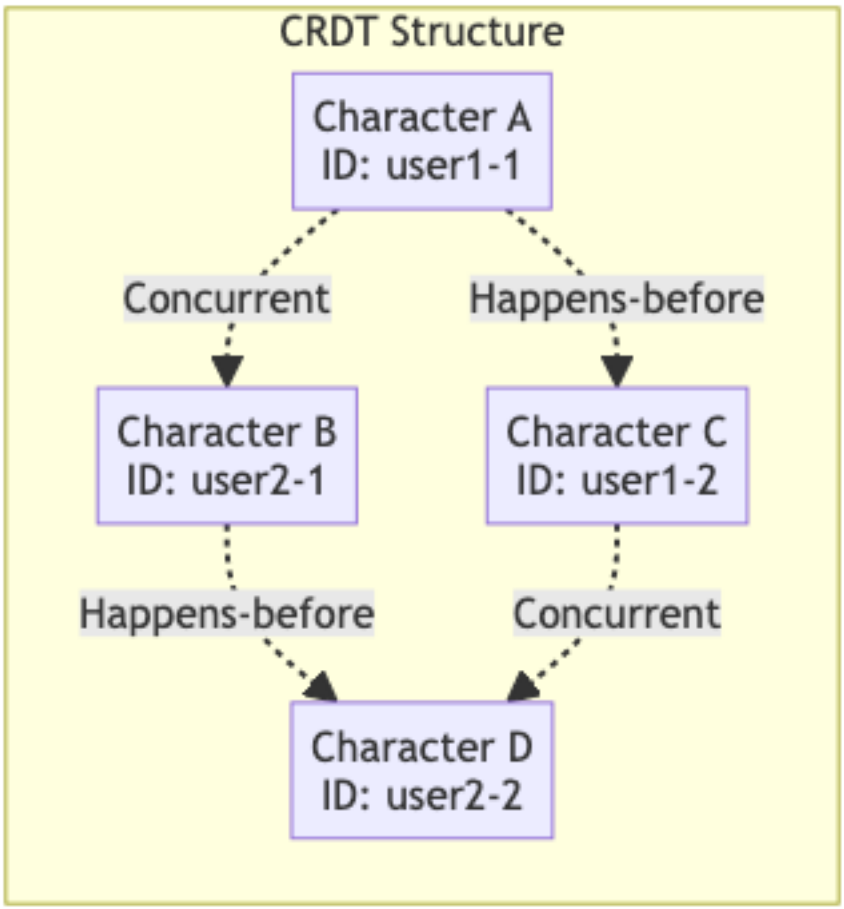
Alternative: CRDT Approach ☐ Back to Top

Core Concept: Conflict-Free Replicated Data Types (CRDTs) are data structures that can be replicated across multiple machines, allowing concurrent updates to be merged automatically without requiring complex transformation logic or a centralized server. Each operation on a CRDT can be applied independently on any replica, and the system guarantees that all replicas will eventually converge to the same consistent state.

Example: CRDT in Action Consider Alice and Bob again, editing a document initially containing “Hello” using a CRDT-based editor.

1. **Alice (Client A)** types ” World” at the end of “Hello”.
   - Alice’s client generates a unique ID for each character and its position (e.g., `(W, ID1, after o)`, `(o, ID2, after ID1)`, `...`).
   - Alice’s client broadcasts these character insertions to other clients.
2. **Bob (Client B)**, concurrently, types ” Awesome” at the beginning of “Hello”.
   - Bob’s client also generates unique IDs for each character and its position (e.g., `(A, IDa, before H)`, `(w, IDb, after IDa)`, `...`).
   - Bob’s client broadcasts these character insertions to other clients.
3. **No central server for transformation:** Each client receives the other’s operations. Since CRDT operations are commutative and associative, they can be applied in any order.
   - When Alice’s client receives Bob’s operations, it simply applies them based on their unique IDs and relative positions. For instance, `(A, IDa, before H)` will be inserted before ‘H’ in “Hello World”.
   - Similarly, when Bob’s client receives Alice’s operations, it inserts them based on their unique IDs and relative positions.

Result: Both Alice and Bob consistently see ” AwesomeHello World”. The CRDT handles the merge automatically because the operations themselves are designed to be conflict-free, ensuring eventual consistency without the need for a central coordination step.



Data Models

☐ Back to Top

Document Structure ☐ Back to Top

```
Document {
  id: UUID
  title: String
  content: OT-Compatible Structure
  metadata: {
    created: DateTime
    modified: DateTime
    version: Integer
  }
  permissions: {
```

```
    owner: UserID
    collaborators: [UserID]
    access_level: Enum
  }
}
```

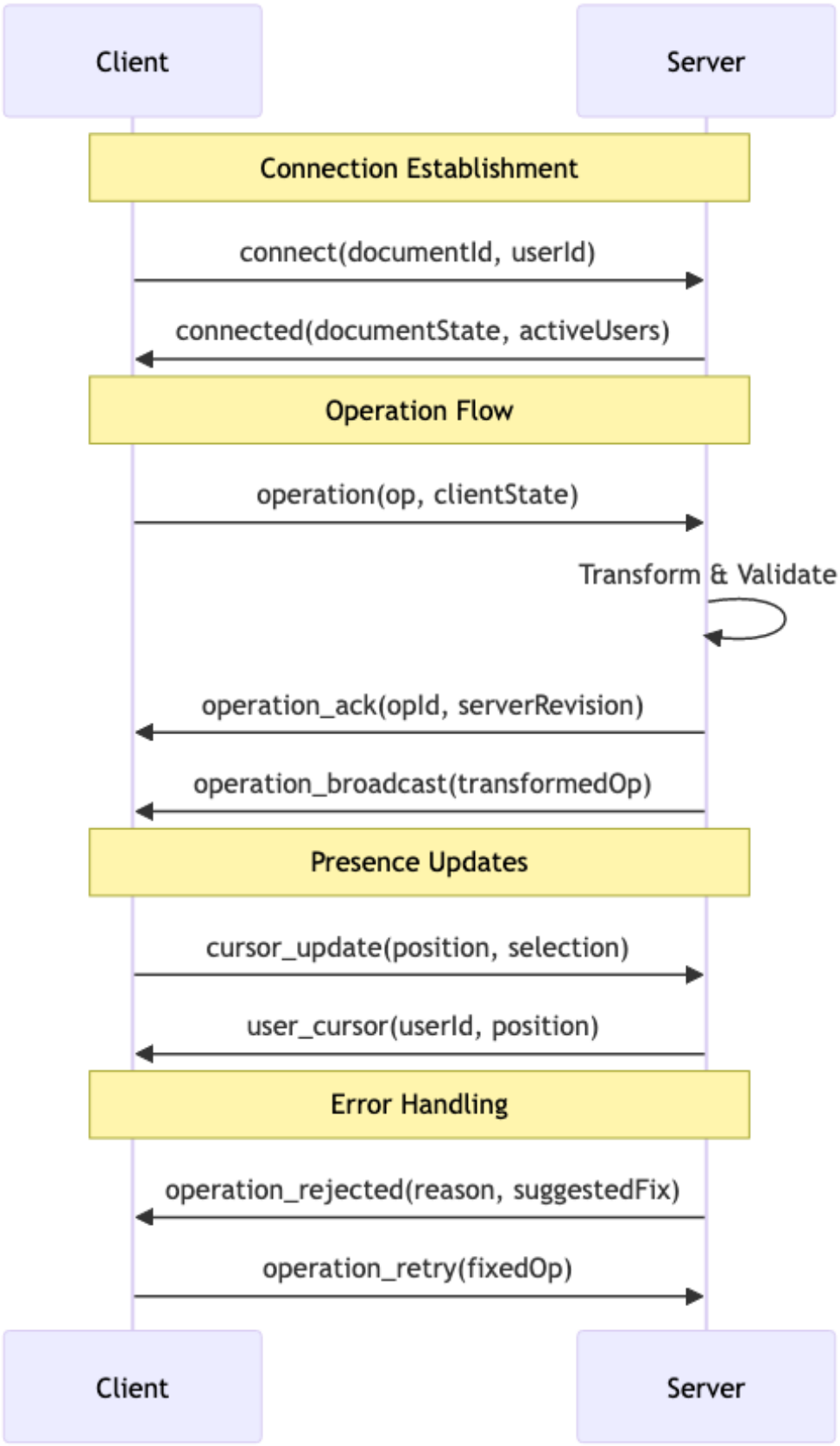
Operation Structure ☐ Back to Top

```
Operation {
  id: UUID
  type: 'insert' | 'delete' | 'format'
  position: Integer
  content: String?
  attributes: Object?
  author: UserID
  timestamp: DateTime
  client_id: String
  version: Integer
}
```

API Design

☐ Back to Top

WebSocket Event Protocol ☐ Back to Top


Client
Server
Connection Establishment
connect(documentId, userId)
connected(documentState, activeUsers)
Operation Flow
operation(op, clientState)
Transform & Validate
operation_ack(opId, serverRevision)
operation_broadcast(transformedOp)
Presence Updates
cursor_update(position, selection)
user_cursor(userId, position)
Error Handling
operation_rejected(reason, suggestedFix)
operation_retry(fixedOp)
Client
Server

REST API Endpoints ☐ Back to Top

- `GET /documents/:id` - Fetch document
- `POST /documents` - Create document

- `PUT /documents/:id/share` - Share document
- `GET /documents/:id/history` - Version history
- `POST /documents/:id/comments` - Add comment

TypeScript Interfaces & Component Props

☐ Back to Top

Core Data Interfaces

```typescript
interface DocumentState {
  id: string;
  title: string;
  content: EditorState;
  collaborators: User[];
  operations: Operation[];
  version: number;
  lastModified: Date;
}

interface Operation {
  id: string;
  type: 'insert' | 'delete' | 'format' | 'retain';
  position: number;
  content?: string;
  attributes?: Record<string, any>;
  author: string;
  timestamp: Date;
  clientId: string;
}

interface User {
  id: string;
  name: string;
  email: string;
  avatar?: string;
  color: string;
```

```
  cursor?: CursorPosition;
  isOnline: boolean;
}

interface CursorPosition {
  line: number;
  column: number;
  selection?: {
    start: number;
    end: number;
  };
}
```

Component Props Interfaces

```
interface EditorProps {
  documentId: string;
  initialContent?: EditorState;
  readOnly?: boolean;
  placeholder?: string;
  theme?: 'light' | 'dark';
  onDocumentChange?: (doc: DocumentState) => void;
  onCollaboratorJoin?: (user: User) => void;
  onError?: (error: Error) => void;
}

interface ToolbarProps {
  editorState: EditorState;
  onCommand: (command: string, value?: any) => void;
  disabled?: boolean;
  customButtons?: ToolbarButton[];
}

interface CollaborationPanelProps {
  users: User[];
  comments: Comment[];
  onInviteUser?: (email: string) => void;
  onAddComment?: (comment: CommentData) => void;
  showPresence?: boolean;
}
```

API Reference

☐ Back to Top

Document Management

- `GET /api/documents` - List user's documents with pagination
- `POST /api/documents` - Create new collaborative document
- `GET /api/documents/:id` - Fetch document content and metadata
- `PUT /api/documents/:id` - Update document title or settings
- `DELETE /api/documents/:id` - Delete document and all operations

Real-time Collaboration

- `WS /api/documents/:id/connect` - Establish WebSocket for real-time collaboration
- `POST /api/documents/:id/operations` - Submit operation for transformation
- `GET /api/documents/:id/operations` - Fetch operation history with pagination
- `POST /api/documents/:id/cursor` - Update user cursor position

Sharing & Permissions

- `POST /api/documents/:id/share` - Generate shareable link with permissions
- `PUT /api/documents/:id/permissions` - Update document access permissions
- `GET /api/documents/:id/collaborators` - List document collaborators
- `DELETE /api/documents/:id/collaborators/:userId` - Remove collaborator access

Comments & Reviews

- `POST /api/documents/:id/comments` - Add comment to specific document position
- `GET /api/documents/:id/comments` - Fetch comments with thread support
- `PUT /api/comments/:commentId` - Update or resolve comment
- `DELETE /api/comments/:commentId` - Delete comment (author only)

Version History

- `GET /api/documents/:id/versions` - List document version snapshots
- `GET /api/documents/:id/versions/:versionId` - Fetch specific version content
- `POST /api/documents/:id/restore/:versionId` - Restore document to previous version

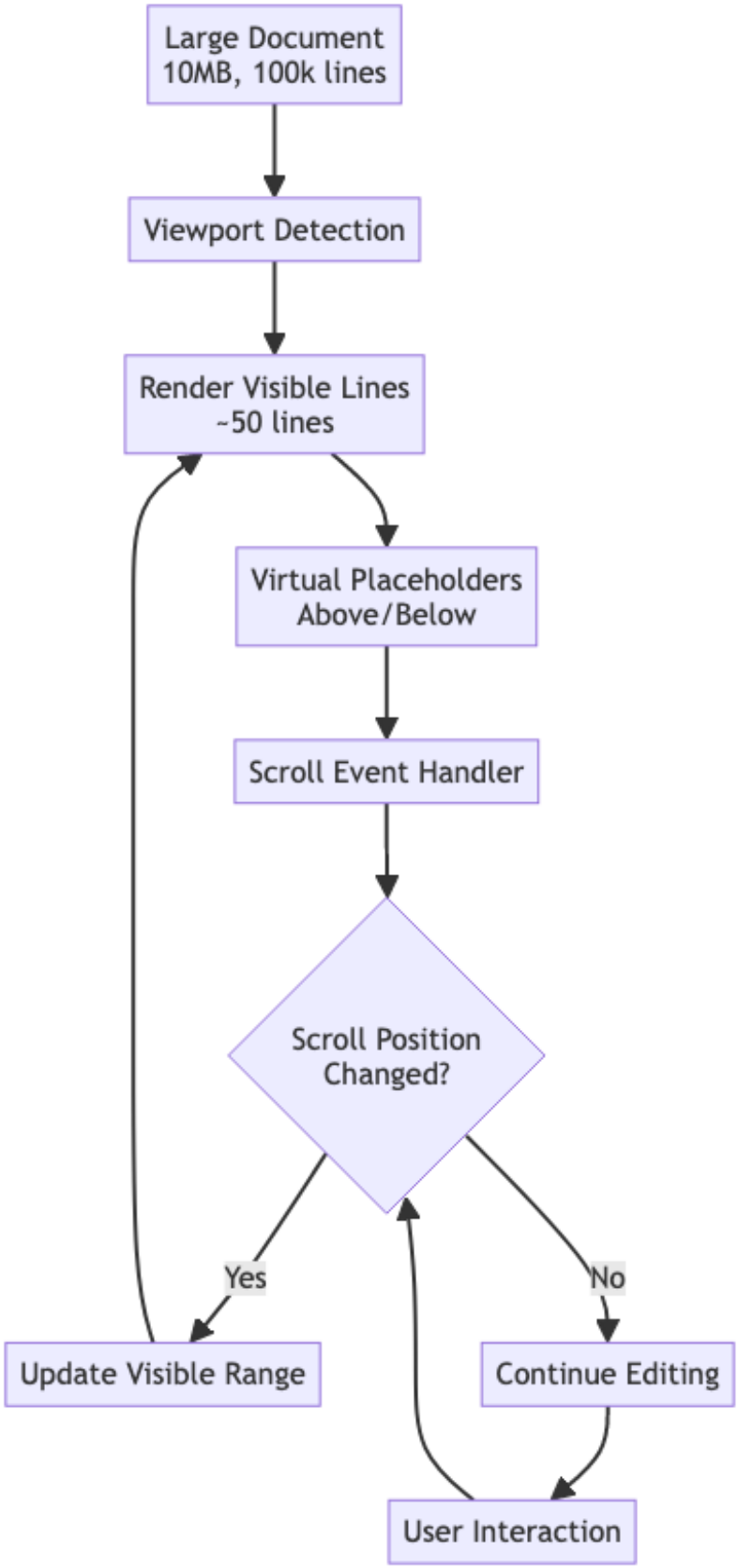
Performance and Scalability

☐ Back to Top

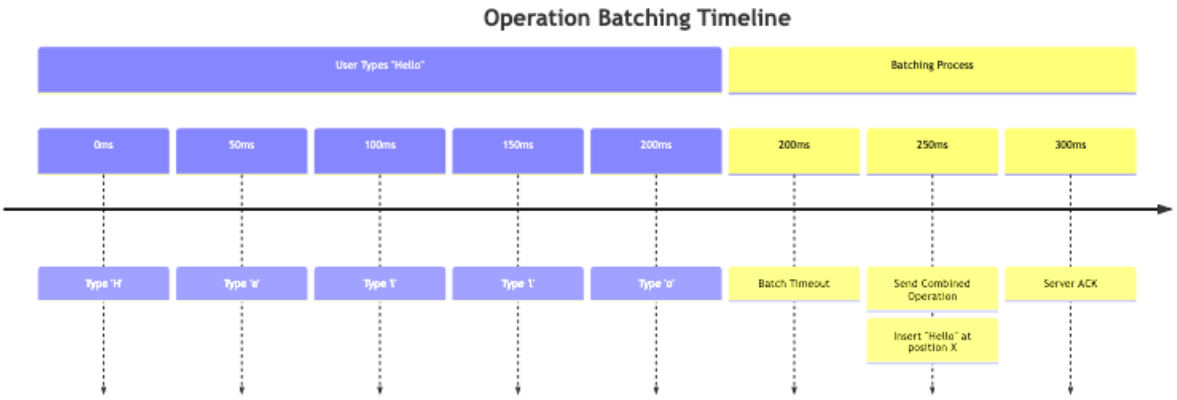
Client-Side Optimizations

Back to Top

Virtual Scrolling for Large Documents Back to Top


Large Document
10MB, 100k lines
Viewport Detection
Render Visible Lines
~50 lines
Virtual Placeholders
Above/Below
Scroll Event Handler
Scroll Position
Changed?
Yes
No
Update Visible Range
Continue Editing
User Interaction

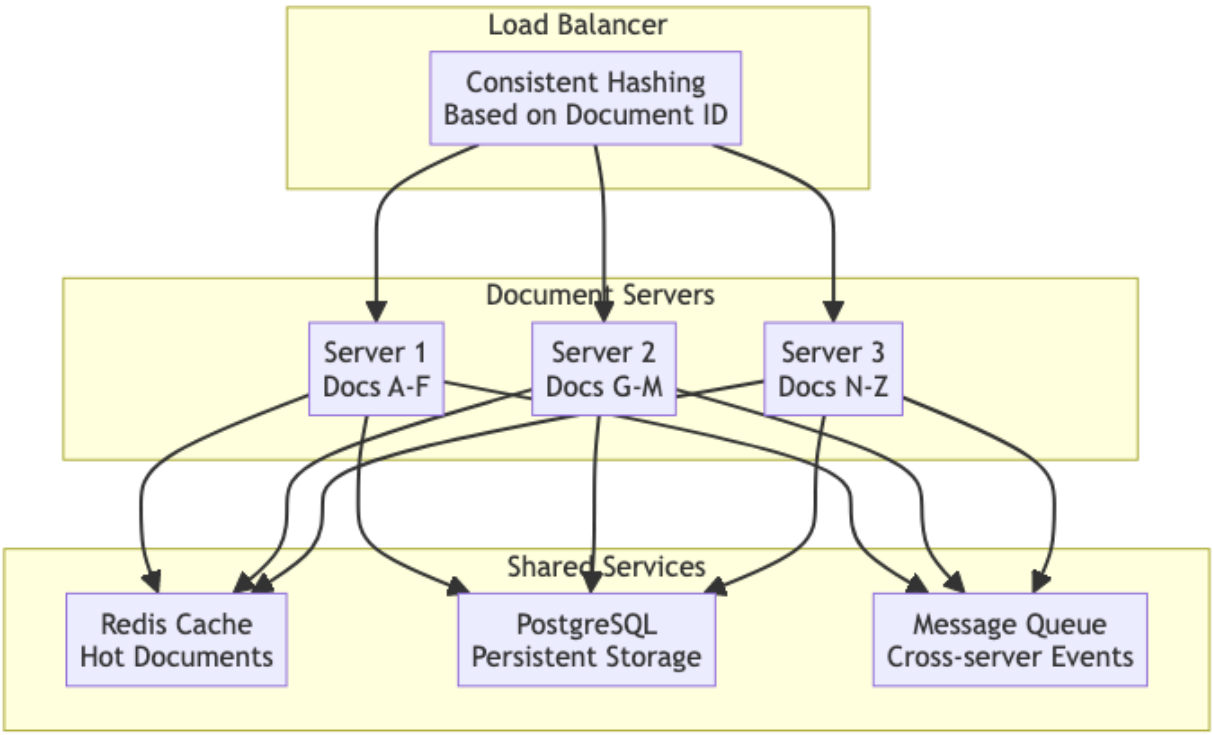
Operation Batching Strategy ☐ Back to Top



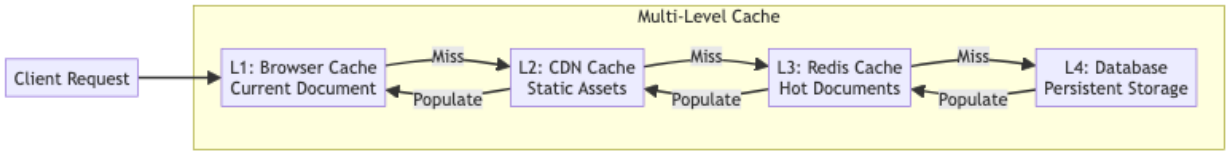
Server-Side Scaling

☐ Back to Top

Document Sharding Strategy ☐ Back to Top



Caching Architecture ☐ Back to Top

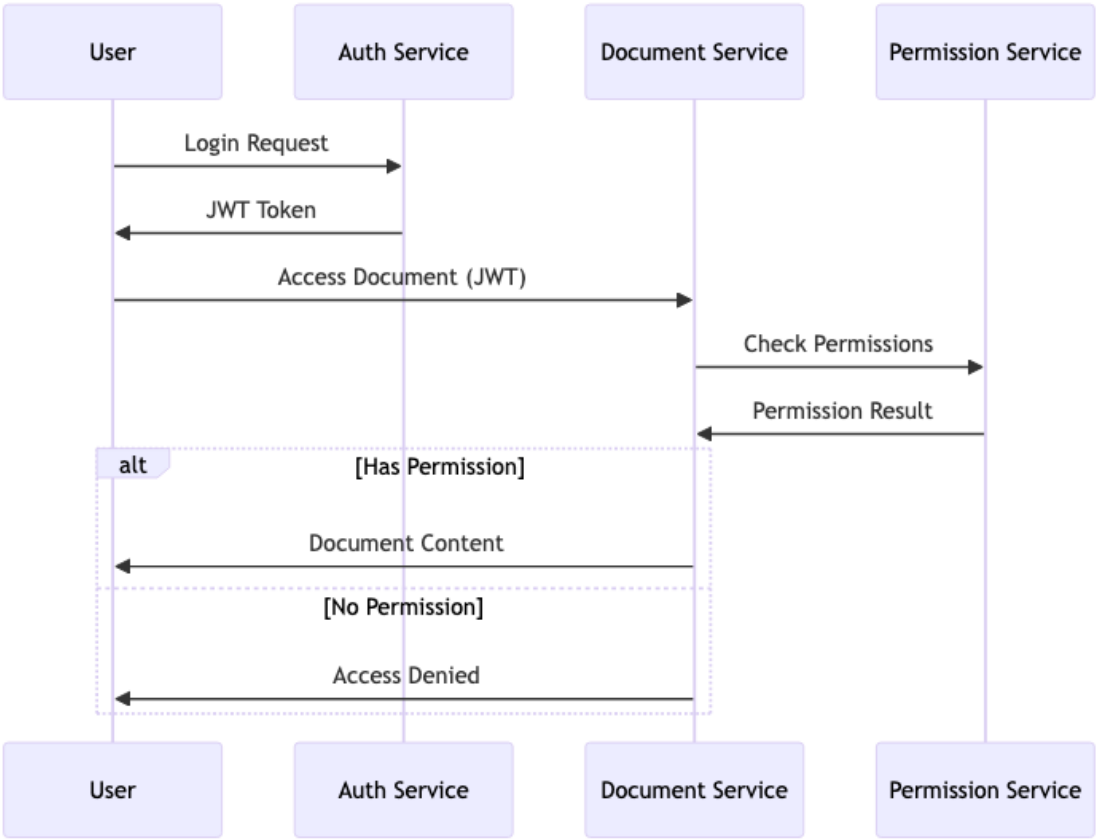


Security and Privacy

☐ Back to Top

Authentication & Authorization Flow

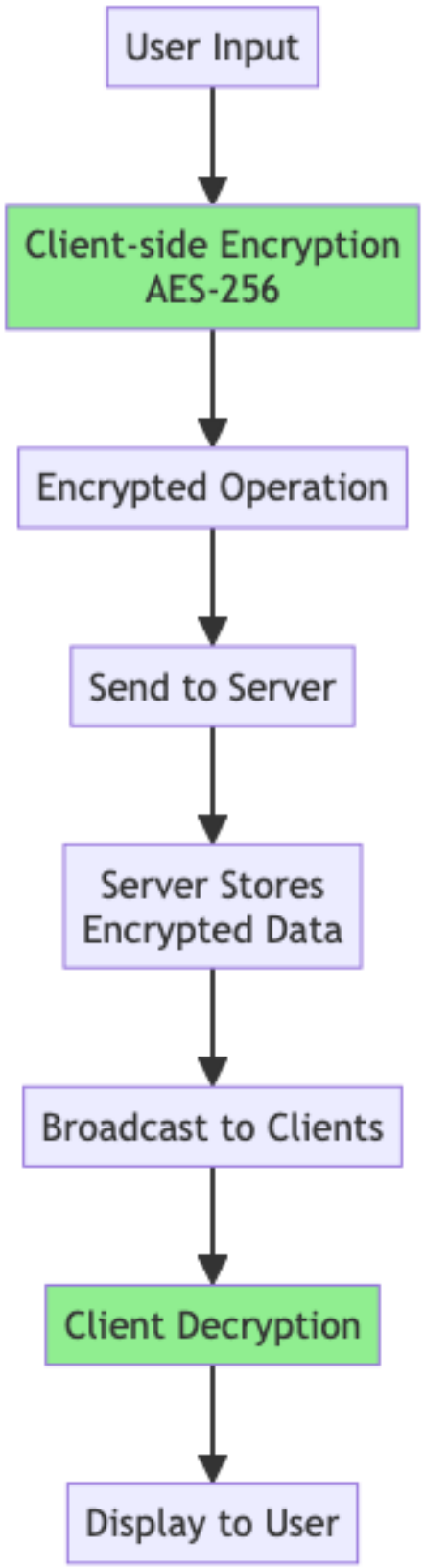
☐ Back to Top



Data Protection Strategy

Back to Top

End-to-End Encryption Flow Back to Top


User Input
Client-side Encryption AES-256
Encrypted Operation
Send to Server
Server Stores Encrypted Data
Broadcast to Clients
Client Decryption
Display to User

Input Sanitization Pipeline

Back to Top

Testing, Monitoring, and Maintainability

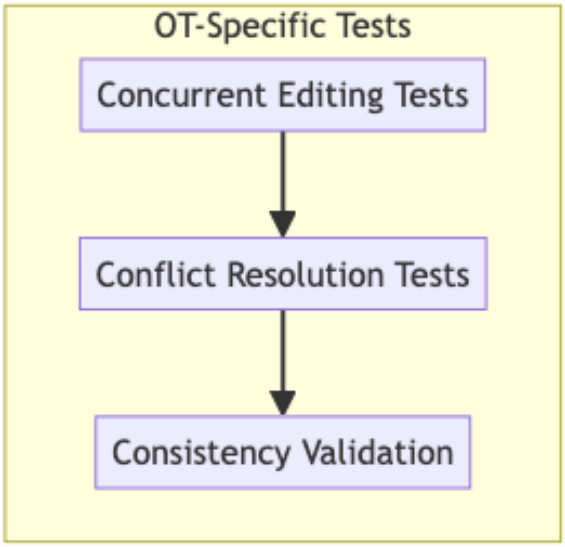
Back to Top

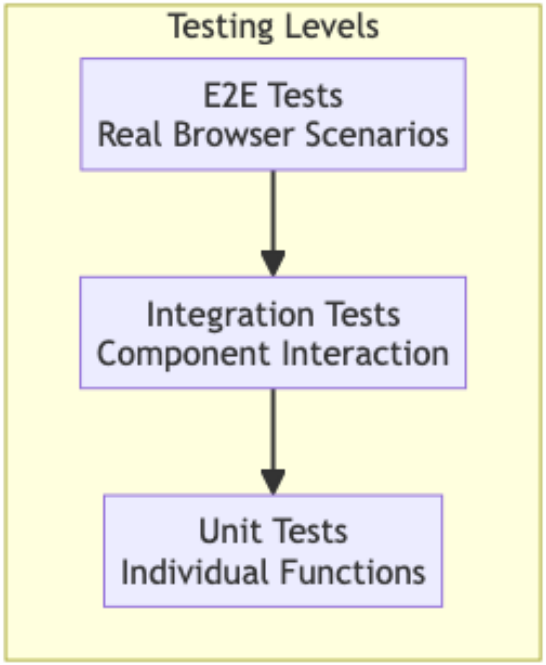
Testing Strategy

Back to Top

Testing Pyramid

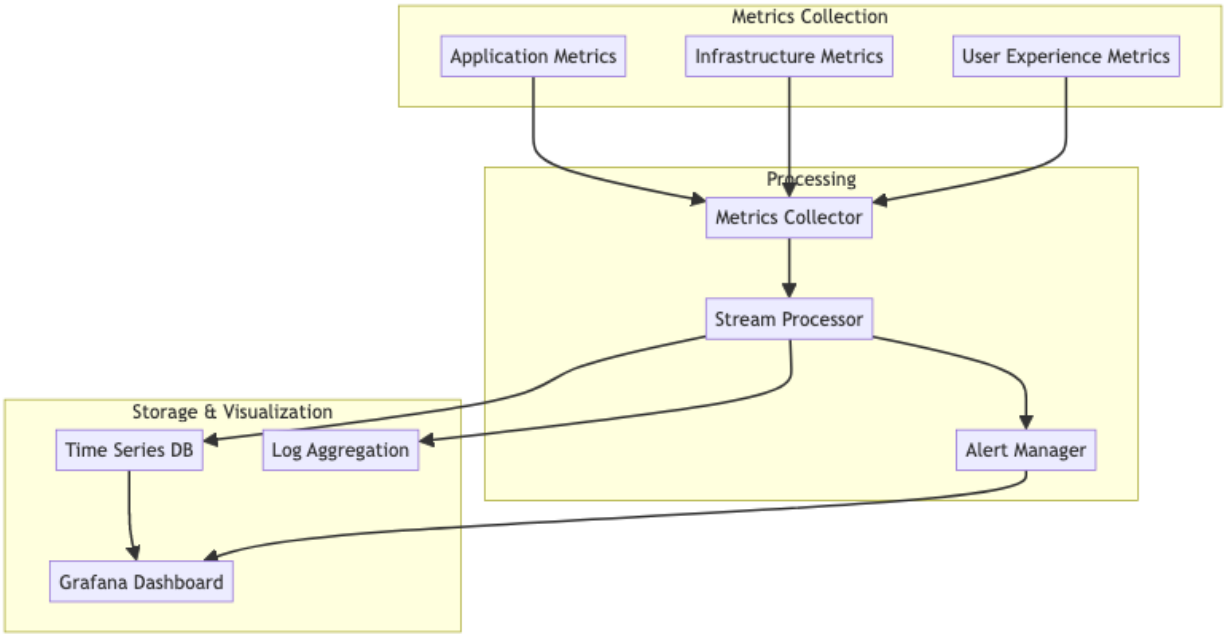
Back to Top





Monitoring Architecture

Back to Top



Key Metrics to Monitor

Back to Top

---

1. **Performance Metrics**:
   - Operation latency (P50, P95, P99)
   - Document load time
   - WebSocket connection success rate
2. **Collaboration Metrics**:
   - Concurrent users per document
   - Operation conflicts per minute
   - Conflict resolution time
3. **System Health**:
   - Server response time
   - Database query performance
   - Cache hit rates

Trade-offs, Deep Dives, and Extensions

Back to Top

OT vs CRDT Comparison

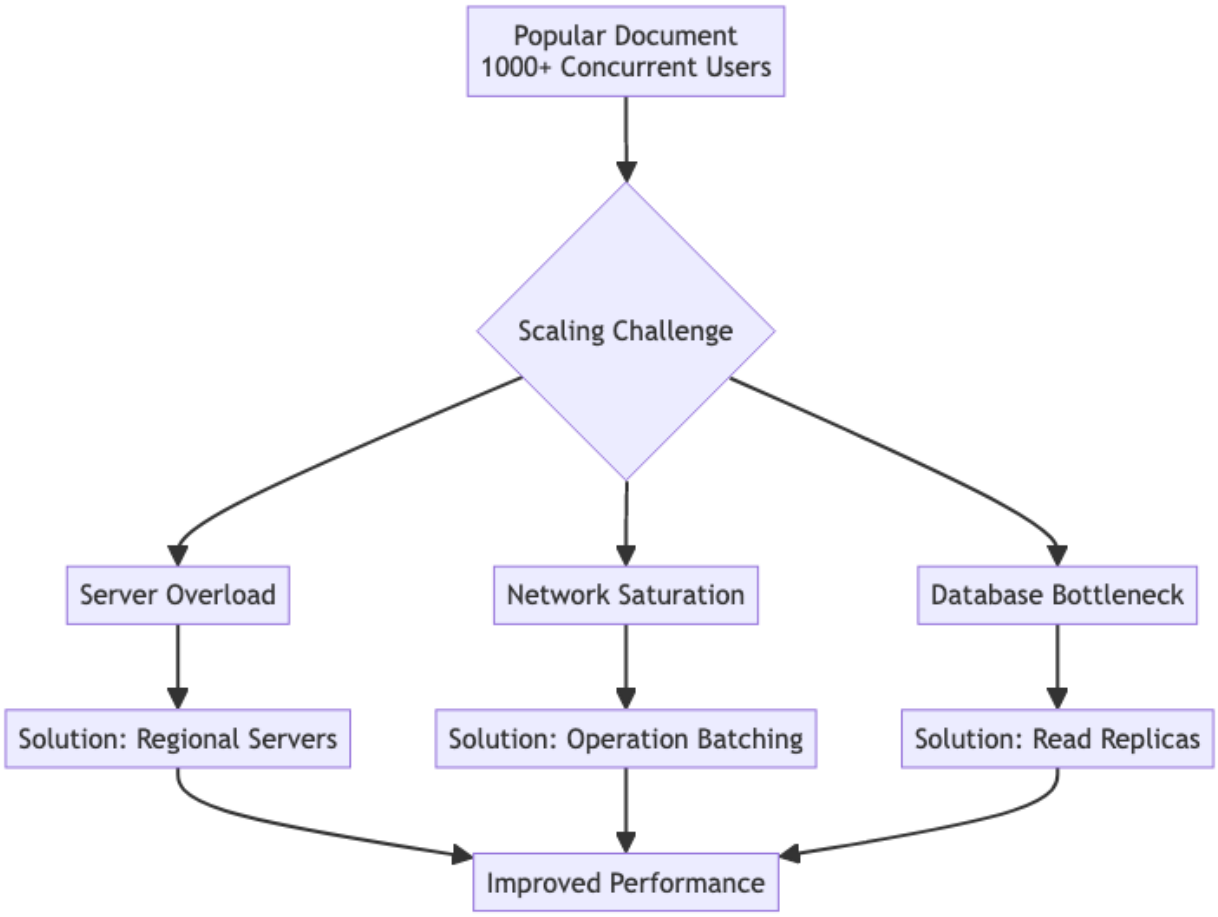
Back to Top

Aspect	Operational Transform	CRDT
Complexity	High implementation complexity	Simpler to implement
Performance	Smaller operation size	Larger data structures
Scalability	Requires central coordination	Fully decentralized
Consistency	Strong consistency	Eventual consistency
Undo/Redo	Complex but possible	Very difficult
Use Case	Text editing, precise control	Distributed systems

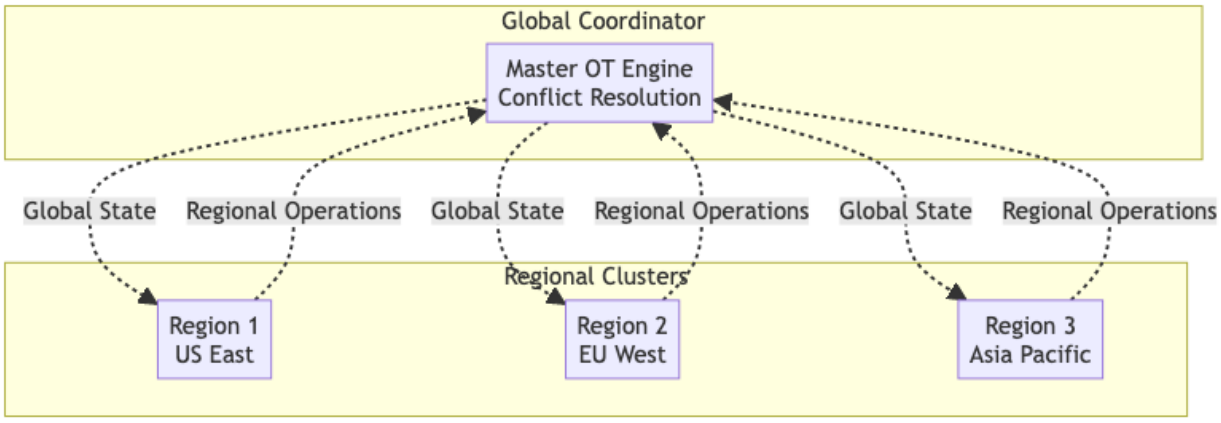
Scalability Bottlenecks & Solutions

Back to Top

Problem: Hot Document Scaling Back to Top



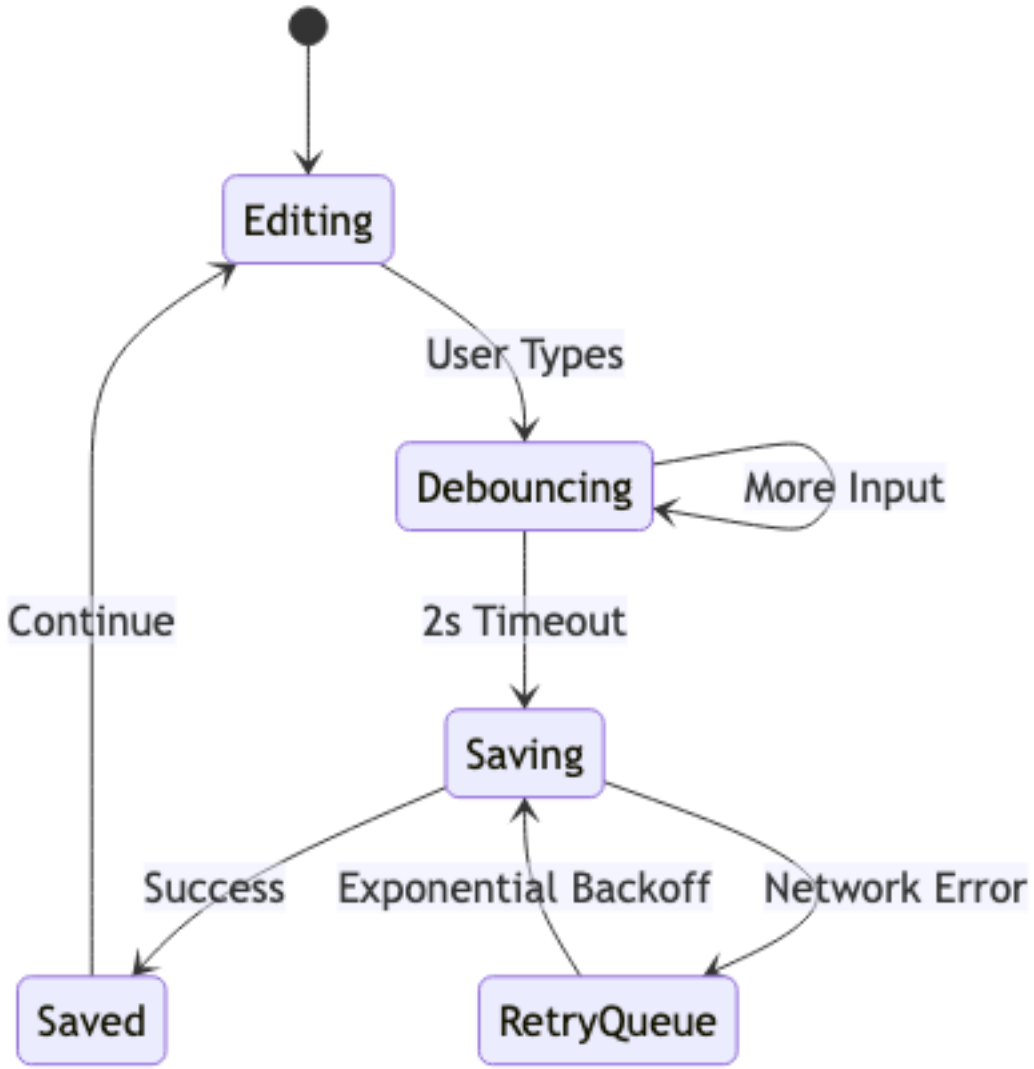
Solution: Hierarchical OT ☐ Back to Top



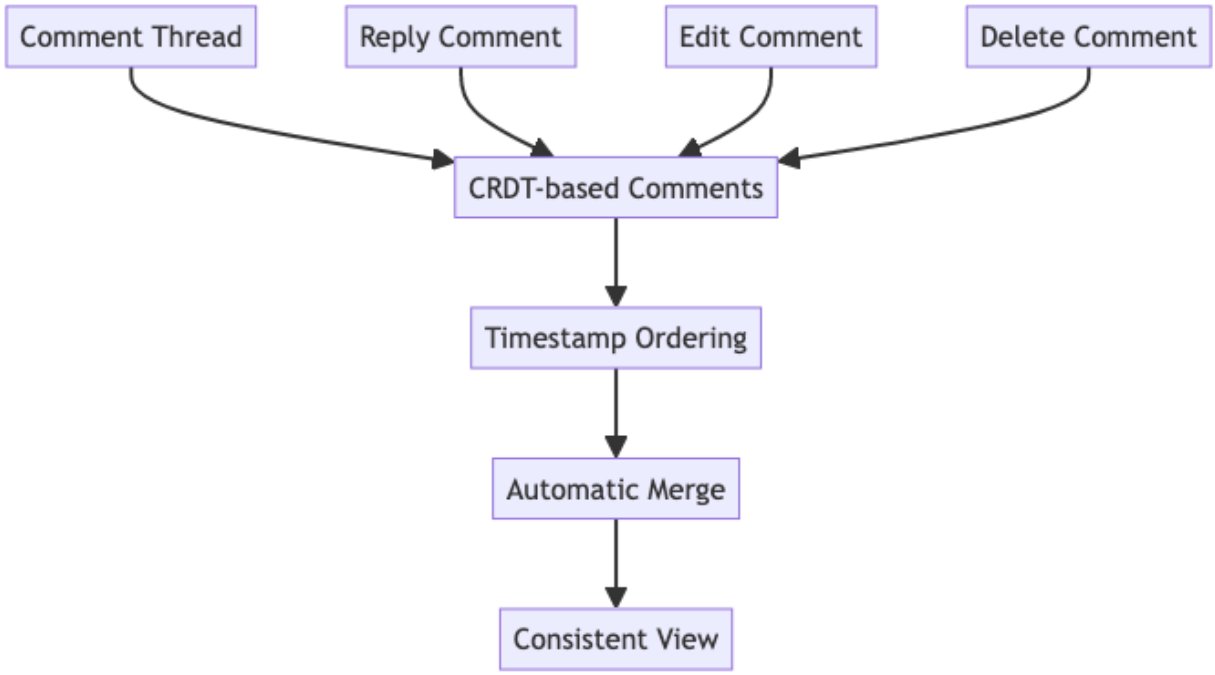
Advanced Features

☐ Back to Top

Smart Auto-Save Strategy ☐ Back to Top



Conflict-Free Comment System ☐ Back to Top



Future Extensions

☐ Back to Top

---

1. **AI-Powered Features**:
   - Smart auto-completion
   - Grammar suggestions
   - Content generation
   - Real-time translation
2. **Enhanced Collaboration**:
   - Voice/video integration
   - Advanced presence indicators
   - Cross-document linking
   - Team workspace management
3. **Performance Innovations**:
   - WebAssembly OT engine
   - Edge computing for regional sync
   - Machine learning for conflict prediction
   - Adaptive quality based on network conditions

This design provides a comprehensive foundation for building a production-ready collaborative text editor with focus on the architectural decisions, algorithms, and system design principles rather than implementation details.