Core Features

## Real-Time Collaboration

Users frequently mention the need for seamless real-time collaboration. This includes:

* Simultaneous editing without conflicts
* Ability to see other users' cursors and edits in real-time
* Built-in chat or commenting system for discussing changes

Source: <https://www.reddit.com/r/Markdown/comments/1ensqfn/collaborative_markdown_with_lix_change_control/>

## Version Control and History

Many users express a desire for robust version control features:

* Ability to track changes and revert to previous versions
* Branching and merging capabilities similar to Git
* Diff view to compare different versions of the document

Source: <https://www.reddit.com/r/Markdown/comments/1ensqfn/collaborative_markdown_with_lix_change_control/>

## Advanced Markdown Support

Users often request support for extended Markdown features:

* Tables with advanced formatting options
* Footnotes and citations
* Task lists and checkboxes
* LaTeX support for mathematical equations

Source: <https://discuss.write.as/t/markdown-inconsistencies-and-markdown-flavour/750>

## Cross-Platform Synchronization

The ability to access and edit documents across different devices is crucial:

* Cloud storage integration (Google Drive, Dropbox, etc.)
* Offline editing capabilities
* Mobile app support

Source: <https://pdf.wondershare.com/create-pdf/online-markdown-editor.html>

## Customization and Extensibility

Users appreciate the ability to tailor the editor to their needs:

* Custom CSS for styling the preview
* Support for plugins or extensions
* Configurable keyboard shortcuts

Source: <https://swimm.io/learn/code-documentation/markdown-editors-key-features-and-8-tools-to-know-in-2024>

## Export and Integration Options

The ability to easily share and publish content is important:

* Export to various formats (PDF, HTML, etc.)
* Direct publishing to blogging platforms
* Integration with version control systems like GitHub

Source: <https://pdf.wondershare.com/create-pdf/online-markdown-editor.html>

# **MarkFlow: Collaborative Markdown Editor**

## **Product Requirements Document (PRD)**

### **1. Project Overview**

## **1.1 Product Description**

SyncMark is a web-based markdown editor with real-time collaboration features, designed to provide a seamless and distraction-free writing experience for individuals and teams. The application offers advanced markdown support, version control, cross-platform synchronization, and extensive customization options.

## **1.2 Target Audience**

* Content creators and writers
* Software developers and technical writers
* Teams collaborating on documentation
* Students and researchers

## **1.3 Key Features (MVP)**

## **3.1 Real-time Collaboration**

Implementation Steps:

1. Choose a real-time collaboration algorithm:
   * Implement Operational Transformation (OT) or Conflict-free Replicated Data Type (CRDT)
   * Recommended: Use Y.js, a CRDT framework that works well with React and Next.js
2. Set up WebSocket connection:
   * Use Next.js API routes with Socket.io for real-time communication
   * Implement connection handling and room management for document collaboration
3. Develop the collaborative editor component:
   * Integrate a rich text editor like ProseMirror or CodeMirror
   * Wrap the editor with Y.js bindings for real-time updates
4. Implement cursor and selection tracking:
   * Use Y.js awareness feature to broadcast cursor positions and selections
   * Develop a custom React component to render collaborator cursors and selections
5. Create a collaborator presence system:
   * Maintain a list of active users in the document
   * Implement a sidebar component to display active collaborators
6. Develop access control:
   * Design a permission system (view, edit, comment)
   * Integrate with the authentication system (Clerk) to enforce permissions

Considerations:

* Ensure low latency for real-time updates (consider using a CDN for WebSocket connections)
* Implement conflict resolution strategies for simultaneous edits
* Handle offline scenarios and reconnection gracefully

## **3.2 Version Control and History**

Implementation Steps:

1. Design the document versioning system:
   * Create a versions collection in MongoDB to store document revisions
   * Implement a versioning strategy (e.g., incremental versions, or timestamp-based)
2. Develop the version creation mechanism:
   * Set up automatic version creation at regular intervals (e.g., every 5 minutes)
   * Implement manual version creation (save points) triggered by users
3. Create a visual diff tool:
   * Use a diff algorithm like google-diff-match-patch
   * Develop a React component to display side-by-side or inline diffs
4. Implement version reversion:
   * Create an API endpoint to revert the document to a specific version
   * Develop the UI for selecting and confirming version reversion
5. Build a timeline view:
   * Design and implement a component to visualize the document's revision history
   * Include metadata like timestamp, author, and version notes

Considerations:

* Optimize storage by implementing compression for version data
* Consider using a separate database or storage solution for version history to prevent performance issues with the main database
* Implement access control for version history to respect document permissions

## **3.3 Advanced Markdown Support**

Implementation Steps:

1. Choose and integrate a Markdown parser:
   * Recommended: Use remark for parsing and rendering Markdown
   * Extend remark with plugins for GitHub Flavored Markdown support
2. Implement syntax highlighting:
   * Integrate a syntax highlighting library like Prism.js
   * Create custom React components for rendered code blocks
3. Develop the live preview pane:
   * Create a split-pane layout with the editor and preview
   * Implement real-time rendering of the preview as the user types
4. Add support for advanced Markdown features:
   * Integrate remark-gfm for tables and task lists
   * Use mermaid.js for diagram support
   * Implement custom React components for rendering these elements
5. Create a toolbar for common Markdown actions:
   * Develop buttons for inserting headers, lists, links, etc.
   * Implement keyboard shortcuts for these actions

Considerations:

* Ensure the preview pane accurately reflects the final rendered output
* Optimize rendering performance for large documents
* Provide options to customize the Markdown rendering (e.g., allowing or disallowing certain HTML tags)

## **3.4 Cross-platform Synchronization (GitHub Integration)**

Implementation Steps:

1. Set up GitHub OAuth integration:
   * Register your application with GitHub
   * Implement the OAuth flow using Clerk's OAuth providers feature
2. Develop GitHub repository connection:
   * Create UI for users to select repositories and files
   * Implement API endpoints to list, read, and write files in GitHub repositories
3. Implement the sync mechanism:
   * Develop a conflict resolution strategy (e.g., three-way merge)
   * Create background jobs for periodic syncing with GitHub
4. Build the UI for manual sync actions:
   * Implement "Pull from GitHub" and "Push to GitHub" buttons
   * Develop a conflict resolution interface for manual merging
5. Handle offline changes:
   * Implement local storage or IndexedDB for offline editing
   * Develop a queue system for syncing changes when connection is restored

Considerations:

* Respect GitHub API rate limits
* Implement proper error handling for network issues or GitHub service disruptions
* Ensure security by storing GitHub access tokens securely (use Clerk's encrypted metadata feature)

## **3.5 Customization (Themes and Syntax Highlighting)**

Implementation Steps:

1. Develop a theming system:
   * Create a set of CSS variables for colors, fonts, and layout properties
   * Implement light and dark themes as the base options
2. Build a theme customization interface:
   * Develop a theme editor component with color pickers and font selectors
   * Implement live preview of theme changes
3. Customize syntax highlighting:
   * Extend the syntax highlighting library to use theme colors
   * Create a separate set of theme options for code blocks
4. Implement theme persistence:
   * Store user theme preferences in the database
   * Apply the user's theme on login across devices
5. Develop layout customization options:
   * Create options for adjusting editor width, line height, etc.
   * Implement a layout editor component

Considerations:

* Ensure all components respect the theming system for consistency
* Optimize theme switching to prevent flickering or slow transitions
* Provide a way to reset to default themes

## **3.6 Extensive Export Options**

Implementation Steps:

1. Implement Markdown to HTML conversion:
   * Use remark-html to convert Markdown to HTML
   * Develop custom plugins for handling advanced Markdown features
2. Create PDF export functionality:
   * Use a library like react-pdf or jsPDF for PDF generation
   * Implement custom styling options for PDF output
3. Develop additional export formats:
   * Implement export to plain text
   * Consider adding export to other formats like LaTeX or DOCX
4. Build an export settings interface:
   * Create options for selecting export format, styling, and content inclusion
   * Implement a preview feature for the exported document
5. Implement batch export:
   * Develop a system for selecting multiple documents for export
   * Create a background job system for handling large export tasks

Considerations:

* Ensure exported documents accurately reflect the Markdown rendering
* Optimize performance for exporting large documents or batch exports
* Implement proper error handling and user feedback for export processes

Implementation Timeline:

1. Weeks 1-3: Set up project structure, implement basic Markdown editor
2. Weeks 4-6: Develop real-time collaboration features
3. Weeks 7-8: Implement version control and history
4. Weeks 9-10: Enhance Markdown support and live preview
5. Weeks 11-13: Develop GitHub integration and sync mechanism
6. Weeks 14-15: Implement theming and customization options
7. Weeks 16-17: Create export functionality
8. Weeks 18-20: Testing, bug fixes, and performance optimization

This expanded guide provides more detailed steps and considerations for implementing each core feature of SyncMark. As you progress with development, you may need to adjust these steps based on specific challenges or requirements that arise. Remember to iterate and test each feature thoroughly as you implement it.

### **2. Technical Requirements**

## **2.1 Platform**

* Web application (primary platform for MVP)

## **2.2 Technology Stack**

* Frontend: Next.js 14
* Authentication: Clerk
* API: Next.js API routes (headless)
* Database: MongoDB (suggested based on real-time collaboration needs and scalability)

## **2.3 Third-party Integrations**

* GitHub API (for repository integration)
* Clerk API (for authentication)

### **3. Detailed Feature Specifications**

## **3.1 Real-time Collaboration**

* Implement Operational Transformation (OT) or Conflict-free Replicated Data Type (CRDT) for real-time editing
* Show cursor positions and selections of collaborators
* Provide a list of active collaborators
* Implement access control (view, edit, comment permissions)

## **3.2 Version Control and History**

* Store document revisions in the database
* Implement a visual diff tool to compare versions
* Allow users to revert to previous versions
* Provide a timeline view of document changes

## **3.3 Advanced Markdown Support**

* Support GitHub Flavored Markdown
* Implement syntax highlighting for code blocks
* Provide a live preview pane
* Include support for tables, task lists, and diagrams (e.g., Mermaid)

## **3.4 Cross-platform Synchronization**

* Implement GitHub integration for working with markdown files in repositories
* Develop a sync mechanism to handle conflicts between local and remote changes
* Provide options to push/pull changes to/from GitHub

## **3.5 Customization**

* Implement light and dark themes
* Allow users to customize syntax highlighting colors
* Provide options for adjusting editor layout and font preferences

## **3.6 Export Options**

* Develop export functionality for PDF, HTML, and other common formats
* Implement styling options for exported documents
* Provide batch export for multiple documents

### **4. User Interface and Experience**

## **4.1 Design Principles**

* Minimal and distraction-free interface
* Focus on the writing experience
* Intuitive and accessible controls

## **4.2 Key UI Components**

* Main editor area with markdown input
* Live preview pane (toggleable)
* Collaboration sidebar (showing active users, comments)
* Version history panel
* Settings/preferences modal

## **4.3 Responsive Design**

* Ensure the application is usable on various screen sizes (desktop, tablet, mobile)
* Implement a responsive layout that adapts to different devices

### **5. Data Management**

## **5.1 Database Schema**

* Users collection: Store user profiles and preferences
* Documents collection: Store document content, metadata, and revision history
* Collaborations collection: Manage document sharing and permissions

## **5.2 Data Flow**

1. User authentication (Clerk)
2. Document creation/loading (MongoDB)
3. Real-time updates (WebSocket/Server-Sent Events)
4. Version control (MongoDB)
5. GitHub synchronization (GitHub API)

## **5.3 Data Security**

* Implement end-to-end encryption for document content
* Use secure connections (HTTPS) for all data transfers
* Regularly backup the database

### **6. Performance and Scalability**

## **6.1 Performance Targets**

* Editor response time: < 50ms
* Document load time: < 2 seconds
* Concurrent users per document: up to 10 (for MVP)

## **6.2 Scalability Considerations**

* Implement database sharding for handling large numbers of documents
* Use a caching layer (e.g., Redis) for frequently accessed data
* Design the system to be horizontally scalable (add more servers as needed)

### **7. Security and Compliance**

## **7.1 Authentication and Authorization**

* Implement secure user authentication using Clerk
* Develop a role-based access control system for documents

## **7.2 Data Protection**

* Encrypt sensitive data at rest and in transit
* Implement regular security audits and penetration testing

## **7.3 Compliance**

* Ensure GDPR compliance for handling user data
* Implement data retention and deletion policies

### **8. Testing and Quality Assurance**

## **8.1 Testing Strategies**

* Unit testing for individual components
* Integration testing for API endpoints and database interactions
* End-to-end testing for critical user flows
* Performance testing for real-time collaboration features

## **8.2 Quality Assurance Process**

* Implement a code review process
* Set up continuous integration and continuous deployment (CI/CD) pipelines
* Conduct regular security scans and vulnerability assessments

### **9. Deployment and Maintenance**

## **9.1 Hosting Environment**

* Deploy the application on a cloud platform (e.g., Vercel for Next.js frontend, MongoDB Atlas for database)
* Set up a staging environment for testing before production deployment

## **9.2 Deployment Process**

* Implement a blue-green deployment strategy to minimize downtime
* Use containerization (Docker) for consistent deployments across environments

## **9.3 Monitoring and Maintenance**

* Set up application performance monitoring (e.g., New Relic, Datadog)
* Implement automated error tracking and reporting
* Establish a regular update and maintenance schedule

### **10. Future Considerations**

## **10.1 Potential Features for Future Releases**

* Mobile applications (iOS, Android)
* Offline mode with local-first data architecture
* AI-powered writing assistance and suggestions
* Integration with other popular platforms (e.g., Notion, Confluence)

## **10.2 Scalability Improvements**

* Implement a microservices architecture for better scalability of individual components
* Explore serverless architecture for certain features to improve cost-efficiency and scalability

This PRD provides a comprehensive overview of the SyncMark project. It covers the main features, technical requirements, and considerations for building a robust and scalable markdown editor with real-time collaboration capabilities. As development progresses, this document should be updated to reflect any changes or new decisions made during the implementation process.

## Project Structure

| markflow/  ├── src/  │ ├── app/  │ │ ├── (auth)/  │ │ │ ├── login/  │ │ │ │ └── page.tsx  │ │ │ └── register/  │ │ │ └── page.tsx  │ │ ├── dashboard/  │ │ │ └── page.tsx  │ │ ├── workspace/  │ │ │ ├── [id]/  │ │ │ │ └── page.tsx  │ │ │ └── page.tsx  │ │ ├── note/  │ │ │ ├── [id]/  │ │ │ │ ├── page.tsx  │ │ │ │ └── edit/  │ │ │ │ └── page.tsx  │ │ │ └── new/  │ │ │ └── page.tsx  │ │ ├── search/  │ │ │ └── page.tsx  │ │ ├── settings/  │ │ │ └── page.tsx  │ │ ├── api/  │ │ │ ├── notes/  │ │ │ │ ├── route.ts  │ │ │ │ └── [id]/  │ │ │ │ └── route.ts  │ │ │ ├── workspaces/  │ │ │ │ ├── route.ts  │ │ │ │ └── [id]/  │ │ │ │ └── route.ts  │ │ │ ├── collaborators/  │ │ │ │ └── route.ts  │ │ │ ├── tags/  │ │ │ │ └── route.ts  │ │ │ ├── search/  │ │ │ │ └── route.ts  │ │ │ ├── export/  │ │ │ │ └── route.ts  │ │ │ ├── session/  │ │ │ │ └── route.ts  │ │ │ ├── changes/  │ │ │ │ └── route.ts  │ │ │ └── sync/  │ │ │ └── route.ts  │ │ ├── layout.tsx  │ │ └── page.tsx  │ ├── components/  │ │ ├── layout/  │ │ │ ├── Header.tsx  │ │ │ ├── Sidebar.tsx  │ │ │ └── Footer.tsx  │ │ ├── notes/  │ │ │ ├── NoteEditor.tsx  │ │ │ ├── NoteList.tsx  │ │ │ └── NoteItem.tsx  │ │ ├── workspaces/  │ │ │ ├── WorkspaceList.tsx  │ │ │ └── WorkspaceItem.tsx  │ │ ├── collaboration/  │ │ │ ├── CollaboratorList.tsx  │ │ │ ├── ReatimeEditor.tsx  │ │ │ ├── CollaboratorPresence.tsx  │ │ │ └── CollaboratorItem.tsx  │ │ ├── search/  │ │ │ └── SearchBar.tsx  │ │ ├── tags/  │ │ │ ├── TagList.tsx  │ │ │ └── TagItem.tsx  │ │ └── ui/  │ │ ├── Button.tsx  │ │ ├── Input.tsx  │ │ └── Modal.tsx  │ ├── lib/  │ │ ├── mongodb/  │ │ │ ├── models/  │ │ │ │ ├── User.ts  │ │ │ │ ├── Note.ts  │ │ │ │ ├── Workspace.ts  │ │ │ │ ├── Collaborator.ts  │ │ │ │ ├── Tag.ts  │ │ │ │ ├── Session.ts  │ │ │ │ ├── Change.ts  │ │ │ │ └── Version.ts  │ │ │ ├── schemas/  │ │ │ │ ├── noteSchema.ts  │ │ │ │ ├── workspaceSchema.ts  │ │ │ │ ├── userSchema.ts  │ │ │ │ ├── workspaceSchema.ts  │ │ │ │ ├── collaboratorSchema.ts  │ │ │ │ ├── tagSchema.ts  │ │ │ │ ├── sessionSchema.ts  │ │ │ │ ├── changeSchema.ts  │ │ │ │ └── versionSchema.ts  │ │ │ └── connect.ts  │ │ ├── services/  │ │ │ ├── noteService.ts  │ │ │ ├── workspaceService.ts  │ │ │ ├── collaboratorService.ts  │ │ │ ├── tagService.ts  │ │ │ ├── versionService.ts  │ │ │ ├── searchService.ts  │ │ │ ├── sessionService.ts  │ │ │ ├── changeService.ts  │ │ │ ├── exportService.ts  │ │ │ └── syncService.ts  │ │ ├── utils/  │ │ │ ├── markdownUtils.ts  │ │ │ ├── exportUtils.ts  │ │ │ ├── realtimeUtils.ts  │ │ │ └── searchUtils.ts  │ │ └── hooks/  │ │ ├── useNotes.ts  │ │ ├── useWorkspaces.ts  │ │ └── useCollaborators.ts  │ ├── types/  │ │ ├── note.ts  │ │ ├── workspace.ts  │ │ ├── session.ts  │ │ ├── change.ts  │ │ ├── collaborator.ts  │ │ └── tag.ts  │ ├── styles/  │ │ └── globals.css  │ ├── config/  │ │ ├── constants.ts  │ │ └── dbConfig.ts  │ └── middleware.ts  ├── public/  │ ├── images/  │ └── fonts/  ├── .env  ├── .env.local  ├── next.config.js  ├── tailwind.config.js  ├── postcss.config.js  ├── tsconfig.json  └── package.json |
| --- |
|  |

## **Key Components and Their Purposes**

## src/app/: Contains all the route components and API routes.

## (auth)/: Group for authentication-related pages.

## dashboard/: Main dashboard page after login.

## workspace/: Workspace-related pages.

## note/: Note-related pages, including viewing and editing.

## search/: Search results page.

## settings/: User settings page.

## api/: All API routes for backend functionality.

## src/components/: Reusable React components.

## Organized by feature (notes, workspaces, etc.) and common UI elements.

## src/lib/: Core logic and utilities.

## db/: Database models and connection setup.

## services/: Business logic separated into services.

## utils/: Utility functions.

## hooks/: Custom React hooks for data fetching and state management.

## src/types/: TypeScript type definitions.

## src/styles/: Global styles and Tailwind CSS configuration.

## src/config/: Configuration files and constants.

## src/middleware.ts: Next.js middleware for request/response modifications.

## public/: Static assets like images and fonts.

## prisma/: Prisma ORM configuration (if using Prisma with PostgreSQL).

## **Notes on Structure**

## The structure uses the App Router, with each route having its own directory and page.tsx file.

## API routes are placed in the app/api directory, following Next.js 14 conventions.

## Components are organized by feature to improve maintainability.

## The lib directory contains all the core logic, separated from the UI components.

## TypeScript is used throughout the project for type safety.

## Tailwind CSS is set up for styling.

## Environment variables are used for configuration (.env files).

## Mongoose is included for database ORM

## Data Models

1. User Model:
   * clerkId (String, required, unique)
   * email (String, required, unique)
   * name (String, required)
   * avatar (String)
   * createdAt (Date)
   * lastLogin (Date)
   * activeSessionId (String, for real-time collaboration)
2. Workspace Model:
   * name (String, required)
   * description (String)
   * owner (Reference to User, required)
   * collaborators (Array of {user: Reference to User, role: String})
   * isPersonal (Boolean)
   * createdAt (Date)
   * updatedAt (Date)
3. Note Model:
   * title (String, required)
   * content (String, required)
   * workspace (Reference to Workspace, required)
   * author (Reference to User, required)
   * tags (Array of References to Tag)
   * lastEditedBy (Reference to User)
   * isArchived (Boolean)
   * createdAt (Date)
   * updatedAt (Date)
   * activeCollaborators (Array of References to User, for real-time collaboration)
   * lastEditedAt (Date, for real-time collaboration)
   * version (Number, for version control in real-time collaboration)
4. Tag Model
   * name (String, required)
   * color (String)
   * workspace (Reference to Workspace, required)
   * createdBy (Reference to User, required)
   * createdAt (Date)
   * updatedAt (Date)
5. Collaborator Model
   * user (Reference to User, required)
   * workspace (Reference to Workspace, required)
   * role (String, enum: ['editor', 'viewer'], required)
   * joinedAt (Date, default: current date)
   * lastAccess (Date)
6. Version Model
   * note (Reference to Note, required)
   * content (String, required)
   * versionNumber (Number, required)
   * createdBy (Reference to User, required)
   * createdAt (Date)
   * comment (String)
7. Comment Model
   * note (Reference to Note, required)
   * author (Reference to User, required)
   * content (String, required)
   * createdAt (Date)
   * updatedAt (Date)
   * isResolved (Boolean, default: false)
   * parent (Reference to Comment, for nested comments)
8. Notification Model
   * recipient (Reference to User, required)
   * type (String, enum: ['mention', 'comment', 'share', 'update'], required)
   * content (String, required)
   * relatedNote (Reference to Note)
   * relatedWorkspace (Reference to Workspace)
   * sender (Reference to User)
   * createdAt (Date)
   * readAt (Date)
   * isRead (Boolean, default: false)
9. Session Model:
   * sessionId (String, required, unique)
   * noteId (Reference to Note, required)
   * participants (Array of References to User)
   * startedAt (Date)
   * endedAt (Date)
10. Change Model:
    * sessionId (Reference to Session, required)
    * userId (Reference to User, required)
    * noteId (Reference to Note, required)
    * operation (String, e.g., 'insert', 'delete', 'replace')
    * position (Number)
    * content (String)
    * timestamp (Date)

## 

## Implementation Sequence

Now, let me explain the structure and the sequence for implementing different parts of your app:

1. Project Setup:
   * Initialize the Next.js 14 project using create-next-app
   * Set up TypeScript
   * Install necessary dependencies (react, next, mongoose, y-js, socket.io, etc.)
2. Backend Setup:
   * Set up MongoDB connection in src/lib/mongodb/connect.ts
   * Create all Mongoose schemas and models (User, Workspace, Note, Collaborator, Tag, Version, Session, Change, Comment, Notification)
   * Implement basic API routes for CRUD operations on these models
3. Authentication:
   * Set up Clerk in middleware.ts
   * Implement sign-in and sign-up pages in the (auth) directory
   * Create authentication middleware for API routes
4. Frontend Core:
   * Implement the basic layout in src/app/layout.tsx
   * Create main pages: Home (page.tsx), Dashboard, and Workspace
   * Implement basic routing
5. Workspace and Note Management:
   * Implement Workspace creation and management
   * Create the basic Note editor component
   * Implement note saving and loading using API routes
6. Markdown Editor:
   * Enhance the Note editor with Markdown parsing and rendering
   * Implement basic formatting tools
7. Real-time Collaboration Setup:
   * Set up WebSocket server using Socket.io
   * Implement basic real-time connection in NoteEditor component
   * Create useCollaboration hook for managing real-time updates
8. Conflict-free Real-time Editing:
   * Integrate Yjs for CRDT
   * Implement real-time syncing of note changes
   * Update the Session and Change models as needed
9. Collaboration Features:
   * Implement CollaboratorList and CollaboratorItem components
   * Create CollaboratorPresence component for real-time user presence
   * Implement invitation system for workspace collaboration
10. Version Control:
    * Implement version creation and retrieval in API routes
    * Create useVersionControl hook
    * Implement VersionHistory component
11. Tagging System:
    * Implement tag creation and management
    * Add tagging functionality to notes
12. Search Functionality:
    * Implement search API with filtering options
    * Create SearchBar component and search results page
13. Export Options:
    * Implement export logic in API routes for various formats (PDF, HTML, etc.)
    * Create UI for export options
14. Comments and Notifications:
    * Implement commenting system on notes
    * Create notification system for various events (mentions, comments, shares)
15. UI/UX Improvements:
    * Implement theming system in globals.css
    * Create ThemeToggle component
    * Refine overall application styling and responsiveness
16. Performance Optimization:
    * Implement lazy loading for note list and search results
    * Optimize real-time collaboration for large documents
    * Add caching mechanisms for frequently accessed data
17. Testing:
    * Write unit tests for components and API routes
    * Perform integration testing for critical user flows
    * Conduct thorough cross-browser and device testing
18. Deployment Preparation:
    * Set up environment variables for production
    * Configure error logging and monitoring
    * Set up CI/CD pipeline

This sequence ensures that you build a solid foundation with the backend and data models before moving on to more complex features like real-time collaboration. It also groups related features together, allowing you to focus on one area at a time and reducing the likelihood of conflicts between different parts of the application.

## **Notes on Mongoose Integration**

1. Mongoose Models: Each model (e.g., Note.ts) will import its corresponding schema and create a Mongoose model.
2. Schema Definitions: Schemas (e.g., noteSchema.ts) define the structure of documents in MongoDB collections.
3. Database Connection: connect.ts will handle the MongoDB connection using Mongoose, possibly with connection pooling for better performance.
4. Services: Update services to use Mongoose methods for CRUD operations and queries.
5. API Routes: Ensure API routes import and use Mongoose models and connection.
6. Environment Variables: Update .env files with MongoDB connection strings and other relevant settings.
7. Types: You may need to update TypeScript types to align with Mongoose document types.

## **Implementation Steps**

1. Install Mongoose: npm install mongoose
2. Set up MongoDB connection in src/lib/mongodb/connect.ts
3. Define schemas in the schemas/ directory
4. Create models in the models/ directory
5. Update services to use Mongoose models and methods
6. Modify API routes to work with Mongoose models
7. Update environment variables for MongoDB connection

This structure maintains the overall organization of the Next.js 14 application while integrating Mongoose for MongoDB interactions. It provides a clear separation of database models, schemas, and connection logic, making it easier to manage and scale your MongoDB-based application.