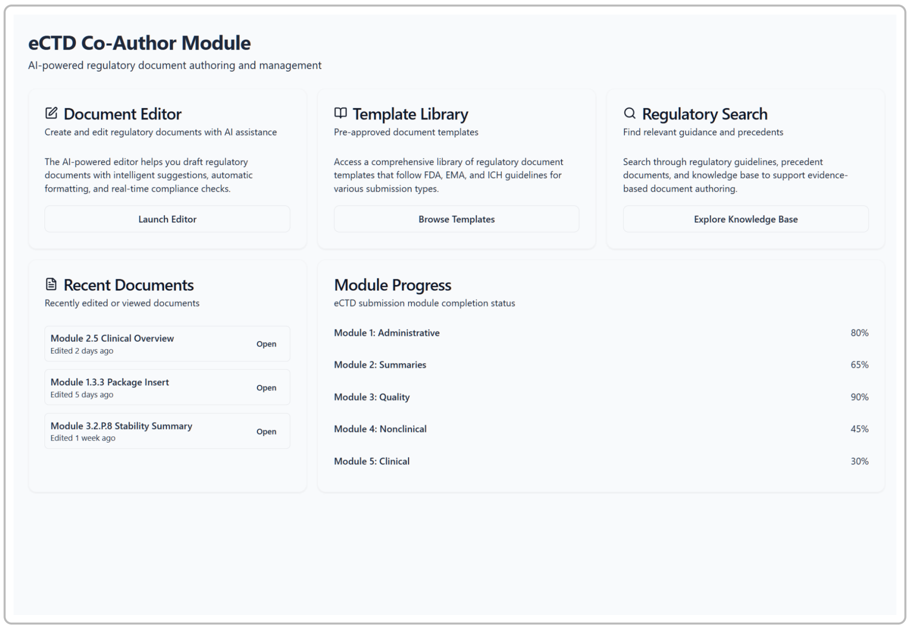
ChatGPT

TrialSage eCTD Co-Author Module: Enterprise-Grade Upgrade Design

Overview: We propose a layered enhancement (not a full rebuild) of the TrialSage eCTD Co-Author Module to introduce advanced authoring features and deep Vault integration. The upgrade builds on the existing interface (see Figure 1) and adds a live collaborative document editor, AI-powered assistance, and robust document management. By integrating a secure "Vault" document repository with real-time editing and compliance tools, this design will streamline regulatory documentation workflows while maintaining compliance with ICH/FDA eCTD guidelines. Key new capabilities include an AI-assisted editor for Modules 1–5, Vault-backed version control with cross-module linking, a regional template library, an intelligent regulatory search engine, detailed module progress tracking, and one-click export to eCTD submission formats. The system will be implemented with a **React** frontend (styled with **Tailwind CSS**), a **Supabase (Postgres)** backend acting as the Vault, and **OpenAI APIs** for language intelligence. Below we outline the upgraded module's features, UI components, backend logic, Vault integration approach, and data model, followed by a visual interface rendering and workflow diagram.



eCTD Co-Author Module
AI-powered regulatory document authoring and management

Document Editor
Create and edit regulatory documents with AI assistance
The AI-powered editor helps you draft regulatory documents with intelligent suggestions, automatic formatting, and real-time compliance checks.
Launch Editor

Template Library
Pre-approved document templates
Access a comprehensive library of regulatory document templates that follow FDA, EMA, and ICH guidelines for various submission types.
Browse Templates

Regulatory Search
Find relevant guidance and precedents
Search through regulatory guidelines, precedent documents, and knowledge base to support evidence-based document authoring.
Explore Knowledge Base

Recent Documents
Recently edited or viewed documents

Module 2.5 Clinical Overview Edited 2 days ago	Open
Module 1.3.3 Package Insert Edited 5 days ago	Open
Module 3.2.P.8 Stability Summary Edited 1 week ago	Open

Module Progress
eCTD submission module completion status

Module 1: Administrative	80%
Module 2: Summaries	65%
Module 3: Quality	90%
Module 4: Nonclinical	45%
Module 5: Clinical	30%

Figure 1: The current TrialSage eCTD Co-Author Module home screen UI. This dashboard provides access to the Document Editor, Template Library, and Regulatory Search, along with recent documents and a Module Progress overview. The upgrade will build on this interface, enhancing each section with new functionality (live editing, templates, search intelligence, etc.) while preserving the user-friendly layout.

AI-Powered Live Document Editor (Modules 1–5)

Feature Summary: At the core of the upgrade is a **live document editor** that allows users to create and edit eCTD submission documents for Modules 1–5 within the browser, with support for importing and exporting **MS Word (.docx)** and **PDF** formats. The editor will include **AI co-authoring tools** to assist with writing and formatting, as well as real-time compliance checks against ICH and FDA guidelines. It will also support collaborative features like revision tracking and document locking for version control. This transforms the current "Launch Editor" feature into a full-fledged web-based word processor tailored for regulatory writing.

Frontend Components & UX: The **Document Editor** is a React component that provides a rich text editing area with a toolbar and side panels for AI assistance and metadata. Key UI elements include:

- **Toolbar with Formatting & Templates:** A toolbar offers standard formatting (styles, lists, tables, etc.) and an "Insert Template" menu to pull in pre-approved content (see Template Library below). Users can also invoke **auto-formatting** commands (e.g. one-click apply of ICH style norms for headings/margins).

- **AI Assistant Sidebar:** An optional sidebar (or popup) powered by OpenAI provides intelligent suggestions. For example, as the user types, the AI can suggest rewordings or completion of sentences. The assistant can also be asked to automatically **format sections** or check for missing content. (E.g. "Ensure this section has all required subsections per FDA guidance.") This uses OpenAI's API to analyze the draft text and return suggestions or identified issues.

- **Real-Time Compliance Checks:** The editor continuously or on-demand runs checks for regulatory compliance rules. Some checks are rule-based (for example, ensuring all required headings for a given document type are present, or flagging if acronyms aren't defined). Others use AI: the content can be sent to an OpenAI endpoint with prompts to verify if it meets certain **ICH/FDA guidelines**. The results are shown as inline warnings or sidebar messages (e.g. "Missing **Clinical Significance** subsection required in Module 2.5 Clinical Overview" or "Formatting of section 1.3.4 does not match eCTD guidance"). This real-time feedback loop helps authors fix issues early. AI-driven compliance monitoring in real-time is an emerging best practice for regulatory writing [1] [2].

- **Collaboration and Revision Tracking:** To support multiple authors and reviewers, the editor will have **track changes** capability. Edits made by a user can be highlighted, and suggestions from the AI or other collaborators can be accepted or rejected. A **commenting** feature lets users discuss sections. Document locking (or check-out) is implemented so only one user at a time can actively edit a given document (others can have read-only access in the meantime). This prevents conflicting edits and aligns with the "version locking" requirement. The UI will show a lock icon and user name if a document is being edited by someone. Once the editor "checks in" the changes (by saving), the lock is released and a new version is created in the Vault.

- **MS Word/PDF Compatibility:** Users can import existing Word documents or PDFs into the editor. The system will use a conversion service or library to transform Word/PDF into HTML for editing. Upon export or save, it can convert the edited content back to .docx and/or PDF. This ensures compatibility with the familiar formats while allowing web-based editing. For PDF inputs (which are not easily editable), the module may instead open them in a viewer and allow annotations or require conversion to Word if editing is needed. For Word files, a library like **docx.js** or a server-side conversion (LibreOffice in headless mode, for example) could be used to import/export content.

Backend Logic: The backend (Supabase/Postgres) manages the document content and versions. When a user opens a document in the editor, the app fetches the latest content from the Vault (either retrieving a stored HTML/JSON representation for editing, or converting a source file to editable format on the fly). As the user writes, periodic auto-saves send updates to the backend (to avoid data loss). On explicit save or check-in, the backend will create a **new version entry** (preserving the previous version for audit trail). The content might be stored as structured data (e.g. Markdown or HTML in the database) and also as a versioned file (docx/pdf) in Supabase Storage. A record might look like: `DocumentVersion(doc_id, version_number, content, author, timestamp, change_summary)`.

To implement **revision tracking**, changes could be computed (e.g. via a diff library) and stored or displayed on the frontend by comparing the current content with the last saved version. The backend can also store the diff or comments for each version. **Version locking** is handled by a flag in the document record (e.g., `is_locked`, `locked_by_user_id`). When a user opens for edit, an API call sets the lock; if someone else tries to open, they get a read-only view or a message that the document is locked.

OpenAI API calls for suggestions/compliance are made via a backend proxy endpoint (to keep API keys secure). For example, a "check compliance" action sends the current text to a server function that calls the OpenAI model with a prompt about verifying against regulations, then returns the result to be displayed.

This AI-powered editor significantly accelerates writing while ensuring accuracy. Industry research indicates that the majority of regulatory document content is derived from existing sources (approximately *50% referencing existing text and 40% reusing text*, with only ~10% truly new) [3] [4]. By integrating dynamic suggestions and compliance checks, the editor helps authors reuse and adapt content correctly, reducing manual effort.

Vault Document Management Integration

Purpose: The "Vault" integration provides robust document management behind the scenes – a secure repository for all regulatory documents with versioning, access control, and audit trails. Instead of building a separate system, we leverage **Supabase (Postgres + Storage)** to act as our Vault. Every document edit, template, or reference is stored and managed in this Vault. This ensures a **single source of truth** for all modules and enables cross-module linking, traceability, and compliance with 21 CFR Part 11 (which mandates audit trails for regulatory documents).

Vault Features and Data Model: Key aspects of the Vault integration include:

- **Secure Storage:** All files (DOCX, PDF, etc.) are stored in a secure cloud storage (Supabase Storage). Access is controlled via user authentication (Supabase Auth) and row-level security rules so that only authorized project members or specific roles can retrieve documents. Sensitive data is encrypted at rest. The Vault ensures documents are not stored on local devices, reducing security risks.

- **Document Versioning:** The Vault keeps a version history for each document. The `Documents` table stores metadata (e.g., document name, module, section, status) and a pointer to the current version, while a related `DocumentVersions` table stores each saved version's details (including file reference in storage, author, timestamp). Users can retrieve older versions or compare differences. This version control allows rollback if needed and provides an audit trail of how a document evolved.

- **Cross-Module Document Linking:** Often a single document might be relevant in multiple contexts (for example, a **Quality Summary** in Module 2 might summarize data elaborated in Module 3, or a **Study Protocol** might be referenced in both Module 1 and Module 5). The system will allow linking documents or sections across modules. In the data model, a `DocumentLink` table can map a source document to a target reference (or we allow a document to have multiple module tags). This way, if one document is updated, any module that references it can be flagged or automatically updated to use the latest version. The UI might allow users to insert cross-references (hyperlinks) to other Vault documents; the system can then manage these links so that upon export, the correct filenames or eCTD cross-reference format is used. (In Veeva Vault systems, similar automated hyperlinking ensures cross-document links remain intact [5] – we would implement a simplified version of this.)

- **Metadata and Tagging:** Each document can be tagged with attributes such as region (US, EU, JP), submission type (IND, NDA, etc.), and keywords. Tags help in organizing and searching documents. The Vault can have a `Tags` table and a many-to-many relation with documents. For example, a document could have tags like "Preclinical, FDA, Final" which can later be used to filter search results or apply template-specific formatting.

- **Audit Trails:** Every significant action is logged. A table `AuditLog` records events like document creation, edits (check-in), approvals, exports, etc., with timestamp, user, and description. This satisfies compliance requirements and allows administrators to trace who did what. Users with proper permission can view an audit trail for a document or project (e.g., "User X created version 2.0 on May 10, 2025 – changed dosing section.").

- **Integration with External DMS (Optional):** While the default Vault is built on Supabase, the architecture can accommodate external Document Management Systems (like Veeva Vault) if needed. For example, instead of storing files in Supabase, API calls could be made to an external Vault to check-in/check-out documents. The design abstracts the storage layer so that in future the Vault could be swapped out or integrated with enterprise EDMS if required. For now, Supabase serves as an all-in-one solution (database + file storage + auth).

By leveraging these Vault capabilities, the platform gains enterprise-grade document management. This brings benefits such as content reusability across submissions and improved collaboration. In fact, using a centralized EDMS/Vault is known to enable *document reuse across submissions, easier navigation between related dossiers, and collaborative authoring with workflow controls* [6]. The Vault ensures that the Co-Author Module isn't just a writing tool, but a controlled repository that supports regulatory requirements throughout the document lifecycle.

Pre-Approved eCTD Template Library

Feature Summary: The Template Library provides a repository of **pre-approved document templates** for common eCTD sections, organized by region (FDA, EMA, PMDA, etc.) and by module. This allows authors to start with standardized structures and boilerplate text that adhere to regional regulatory guidelines. Instead of writing from scratch, users can browse the library, select the appropriate template for the document they need (e.g. "Module 2.5 Clinical Overview – FDA format"), and inject that content into the live editor.

Template Content: Templates will include recommended headings, section outlines, and example content or annotations guiding what to fill in. Each template is essentially a document in the Vault marked as a template. We will create templates covering modules 2–5 summaries, module 3 quality document formats, module 4 nonclinical study report templates, module 5 clinical study report, etc., as well as Module 1 region-specific documents (like FDA's 1.3.4 Financial Certification form, etc.). By having separate templates for FDA vs EMA (for instance, an EMA Module 1 might have different requirements), the library ensures regional compliance from the start. These templates are curated and approved by internal regulatory experts, so users trust their accuracy.

UI and Usage: The Template Library is accessible from the main dashboard ("Browse Templates" button) and also directly from within the editor (via an "Insert Template" toolbar action). The **Template Library UI** is a React component that likely appears as a modal or separate page with a list/tree of templates. Key UI behaviors:

- Users can filter or navigate by region and module. For example, pick "EMA" and "Module 3" to see all EMA-specific Quality document templates.
- Each template entry shows a title and a brief description. For instance, "Module 2.5 Clinical Overview (EMA) – Structure for the Clinical Overview section as per EMA guidelines."
- Clicking a template may show a preview of its content (perhaps in a read-only view) so the user can confirm it's what they want.

From the editor, when inserting a template into an active document, the system will likely merge the template content into the current document. If the document is empty, it simply loads the template. If not empty, the user might choose to append or replace with the template (with a warning to avoid overwriting content unintentionally). The insertion could also be intelligent: for example, if you have a partial draft and you choose a template, it might only insert sections you haven't written yet, or just bring in standard headers.

Data and Backend: Templates are stored in the Vault like other documents, but flagged as templates (e.g., `documents.type = 'template'`). We might organize them by region and module via metadata. A small table or JSON metadata can list available regions and modules for which templates exist. When a user browses or searches in the template UI, the app queries the database (e.g., `SELECT * FROM documents WHERE type='template' AND region='FDA' AND module=2`). The content is likely stored as a file (docx) and as an HTML/JSON for preview. Templates will be maintained by admin users, who can update them as regulations change.

Using templates ensures consistency and saves time. It reduces human error because the structure follows health authority guidelines. Even leading industry tools emphasize template-driven authoring for compliance [7]. By starting from a template, authors are less likely to omit required sections or use incorrect format, and they benefit from boilerplate text that they can modify instead of writing from scratch.

Integrated Regulatory Search Engine (Knowledge & Precedents)

Feature Summary: The upgraded module will include a **Regulatory Search** tool that connects to the Vault and other knowledge bases to let users find relevant guidance, precedent text, and related submissions. This is accessible via the "Regulatory Search" panel (from the dashboard or within the editor). The goal is to enable authors to quickly lookup how similar content was written previously or to find official guidance lines, without leaving the platform.

Use Cases: - An author writing a **Clinical Overview** might search for "previous Phase II clinical overview efficacy section" to see how efficacy results were summarized in an earlier submission. - While drafting, the user could highlight a term or requirement and search for FDA guidance documents or ICH guidelines about that topic. - The tool could fetch **precedent text** – actual paragraphs from prior approved submissions in the Vault – which the author can use as a reference or even adapt (with proper caution to not copy confidential info between different projects, unless it's the same sponsor's data). Because content reuse is common in regulatory writing [8] [9], having quick access to past language can significantly speed up drafting.

UI Integration: The Regulatory Search might be a sidebar in the editor or a modal that shows search input and results. It includes a search bar where users can enter queries (keywords or questions). Results could be categorized by source: e.g., *Guidances* vs *Past Submissions*. Each result item might show a snippet of text with the keyword highlighted and indicate the source document. Users can click a result to view more (perhaps open the full document in a viewer or scroll within a preview pane). If the user wants to use a result, options could include "copy snippet" or "insert into editor," which would paste the selected text into the document (with appropriate citation or formatting if needed).

Data Sources: Primarily, the search will index the Vault documents (text of all prior documents the user has access to). This means every document's content is searchable. We can implement full-text search using Postgres full-text indexing or use Supabase's built-in search capabilities. For smarter search (semantics, not just exact keyword), we could integrate OpenAI's embeddings: e.g., pre-compute embeddings for paragraphs of documents and use vector similarity to find semantically related content to the query. For official guidance, we could load a library of public guideline documents (like ICH M4 guidelines, FDA technical guidance documents) into the Vault or a separate index so they are also searchable. This effectively creates a knowledge base that the search engine queries.

AI Enhancement: We can layer AI on the search in two ways: 1. Use NLP to **interpret queries** (e.g., understand a question about regulations and translate to relevant keywords). 2. Use a generative model to summarize or elaborate on found text. For example, after retrieving some precedent text, an "AI summary" button could generate a concise summary or even adapt the text to the current context (with user verification). However, caution is needed to ensure factual accuracy and not to introduce errors, so primarily the search will retrieve actual text from trusted sources.

Backend Implementation: Whenever a document is saved in the Vault, an asynchronous process can update the search index (e.g., update the full-text index or re-generate embeddings). The search query hits an API endpoint that either leverages Postgres full-text search (with `tsvector`) or calls an external service for semantic search. If using embeddings, we might use the **pgvector** extension in Postgres (which Supabase supports) to store embedding vectors for document sections, and do vector similarity search on the query embedding. The result is a set of document IDs with relevance scores, which the API then returns to the frontend with snippets.

The integrated search engine means authors don't have to leave TrialSage to hunt down references – it's all at their fingertips. Connecting to a Vault of past submissions and guidances supports the content referencing approach that makes regulatory writing more efficient [8]. It also promotes consistency, since writers can align their language with precedent text. This feature, combined with the template library, leverages prior knowledge to **accelerate evidence-based authoring**.

Module Progress Tracking & Validation Dashboard

Feature Summary: The Module Progress section of the dashboard will be upgraded to actively track completion status of each eCTD module (Modules 1 through 5) in real time. Instead of static percentages, it will compute progress based on the presence and status of required documents in each module, and validate the submission content against a completion checklist. Visual status markers will highlight which parts of the submission are done, in progress, or missing, helping project managers ensure nothing is overlooked.

Components and UI: On the main dashboard (Figure 1, bottom-right), each module is listed with a percentage complete. We will enhance this by making the module names clickable to view a detailed breakdown. Upon clicking "Module 3: Quality (90%)" for example, the user might see a pop-up or navigate to a **Module Detail page**. This page would list all the expected sections/documents for Module 3 (based on the eCTD specification and region requirements) with status indicators:

- Each required document (granule) could have an icon: green check for completed, yellow for in progress, red or empty for not started. If a document is present in the Vault and marked final, it's "complete". If a draft exists, "in progress". If nothing, "not started".
- The percentage is calculated as (number of completed documents / total required documents * 100). For example, Module 2 might have 20 required docs, if 13 are done, that's 65%.
- We will allow some customization, e.g., marking certain documents as "not applicable" for a particular submission (then they wouldn't count as required). The system may allow the user to adjust the checklist per project because not all sections apply to every submission type.

Additionally, each module or document line could display a status tag (like *Not Started, Draft, In Review, Approved*). This ties into a simple workflow where, say, once an author finishes a draft, they mark it "In Review", and after QA it becomes "Approved". The progress calculation might consider "Approved" as complete vs just drafted.

Validation against Checklist: We will maintain an internal **eCTD Checklist** (possibly a JSON or database table) that outlines what documents are expected for each module by submission type and region. For example, for an **FDA IND** submission, list all Module 1 docs needed, Module 2 summaries, etc. The system uses this checklist to populate the module list and to validate completeness. If any required element is missing or still in draft, the progress won't hit 100%. This acts as a **quality control** to ensure all pieces of the eCTD are present. If something is missing or status is still "Draft" while nearing submission, the UI might flag it (e.g., highlight the row in red or show a warning "Module 4 has pending items").

We can also integrate basic **technical validation**: for instance, if a PDF is supposed to be OCR'd or under a certain size for eCTD, those could be additional checks. The OpenAI compliance check could also feed into validation – e.g., if AI identified a compliance issue in a document, the module might not count as fully complete until that is resolved.

Backend Logic: The Supabase database will have tables like `ModuleRequirements` (with fields: region, submission_type, module_number, doc_type, required_flag). Another table `ProjectDocs` or the main Documents table links each actual document to a project, module, and maybe a requirement ID if it corresponds exactly. A background job or a trigger on document status change can recalc the completion percentages and store them, or it can be calculated on the fly when the dashboard is loaded (the latter might be simpler initially). Given Postgres, a view or query can compute the counts of completed docs per module easily.

For status tracking, a field in the Documents table (or a separate workflow table) holds the status (Draft/Review/Final). The UI could allow updating this status. When a document's status flips to final/approved, the progress bar updates.

Overall, this feature turns the dashboard into a **real-time submission tracker**, helping teams manage the assembly of the eCTD. By validating against a checklist, it reduces the risk of missing a document last minute. Similar metrics and tracking are often used in regulatory project management to ensure submission readiness [10] [11]. Here, it's built directly into the authoring tool for convenience.

Export to Structured PDF, DOCX, and eCTD XML

Feature Summary: Once documents are authored and reviewed, the system should facilitate easy export of the submission package. The upgraded module will provide **export options** to generate the compiled submission in multiple formats: as a collection of PDFs (with an eCTD XML backbone), as a set of Word documents, or other structured outputs. Essentially, this is the "publishing" step where the authored content is packaged per the eCTD specification for submission to regulatory authorities.

Export Tool UI: On the dashboard, a button (or in a dedicated "Export/Publish" section) allows users to initiate the export. The **Export Tool UI** (React component) might present a form or wizard with options: select region (to apply the correct eCTD DTD schema for the XML), select sequence number (if it's a sequence in a rolling submission), and possibly which modules to include (by default all completed modules). The user can choose the output format: - *"eCTD Package (XML + PDFs)"* – this will produce a set of folders (Modules 1-5) with documents in PDF (since regulatory submissions prefer PDF for text documents), along with the required XML index files. - *"Combined PDF"* – for internal review, maybe generate a single PDF that merges selected documents in order. - *"DOCX files"* – perhaps to download the source Word files for archival or additional editing outside the system.

After choosing, the user triggers the export. The UI shows a progress indicator as it generates the files (this could take a bit of time if conversions are needed). Once ready, a link is provided to download a ZIP file containing the package (for eCTD, the ZIP would have the folder structure and XML).

Backend Export Logic: This is a critical piece that ensures compliance: 1. **Document Conversion:** The backend will take the latest version of each required document from the Vault. If it's in DOCX or HTML form, convert it to PDF (using a conversion service or an API like LibreOffice or Microsoft's conversion if available). If it's already PDF (some may be attachments like literature references), ensure it meets any requirements (e.g. PDF version, no active content). 2. **eCTD XML Generation:** Based on the documents included, generate the XML backbone files. For a given region, the eCTD has an index.xml (and for FDA, a us-regional.xml). We will have a template or library for generating these. Essentially, the code will iterate through modules and documents, and create XML entries with their file names, titles, etc., in the proper hierarchy. For example, if Module 2.5 Clinical Overview is included, we add an `<leaf>` entry in the XML with its file path (like `/m2/25-clin-over.pdf`) and title. This requires knowing the CTD structure which our ModuleRequirements checklist provides. We also assign sequence numbers or use the provided sequence. Using an XML library (like Python's lxml or a Node.js XML builder if using JS on backend) can help create these files according to the ICH spec. 3. **Packing and Delivery:** Once all PDFs and XML files are ready in a temporary storage location (or memory), the backend packages them (if multiple files, a ZIP is created). The file is then made available for download via Supabase Storage or a direct download link. The system can also automatically save this package in the Vault for record-keeping, or update an "Export" record (with timestamp, who exported, etc.).

The export module ensures the final output is **submission-ready**. It essentially automates what a publisher would do manually – collating files and writing the XML table of contents. The design will follow eCTD standards so that the output can be directly uploaded to an agency portal without further modification. This capability, combined with the earlier features, means TrialSage can go from authoring to publishing in one integrated flow. Users get both the working formats (DOCX for editing) and the final formats (PDF/XML for submission) in a consistent, repeatable way.

To summarize the steps in export, for clarity: 1. **Select Export Options** (region, modules, format). 2. **Gather Documents** – query the Vault for all final versions of docs in scope. 3. **Convert to PDF** – for each doc (if not already PDF). 4. **Generate XML** – create eCTD index.xml with entries for each document in the correct module structure. 5. **Package** – bundle the XML and PDFs (plus any other files like study data if those are part of submission). 6. **Deliver** – provide download link or save to Vault.

This process aligns with standard eCTD publishing practices, ensuring compliance with technical requirements [12] [13]. By automating it, we reduce the chance of human error in assembling the submission and greatly speed up the time from final document to actual submission.

Frontend Architecture and Components

The frontend will be organized as a **React application** with multiple interactive components corresponding to the features above. Tailwind CSS will be used for a cohesive, responsive UI design (as already exemplified in the existing interface). Here are the primary components/pages and their roles:

- **Dashboard Page (`Dashboard.jsx`):** This is the entry screen (as shown in Figure 1). It contains sub-components for Recent Documents list and Module Progress overview. It fetches summary data (recent docs the user edited, and progress percentages for each module) from the backend on load. It provides navigation to other features (Editor, Templates, Search, Export). We will maintain the clean card-based layout: e.g., a card for Document Editor (with "Launch Editor" button), card for

Template Library ("Browse Templates"), card for Regulatory Search ("Explore Knowledge Base"), etc., as in the screenshot, but these will now link to the actual new functionalities.

- **Live Document Editor (`Editor.jsx` and children):** This component implements the document editing page. It may be composed of a top navigation (e.g., to return to dashboard or to show document title and status), the main editable area, and side panels. We might break it down further:

- `EditorToolbar.jsx` (holds formatting buttons, template insert dropdown, AI check button, etc.),
- `EditorContent.jsx` (the content editable area, possibly using a rich text library such as ProseMirror, Slate, or Draft.js under the hood to handle robust editing and track changes),
- `AISuggestions.jsx` (sidebar panel listing AI suggestions or showing compliance results),
- `CommentsPanel.jsx` (if implementing commenting),
- `VersionHistoryModal.jsx` (a modal to show past versions from the Vault, with option to restore or compare).

The Editor component communicates with backend via Supabase client or via API endpoints for saving content, retrieving templates (when inserted), and running AI checks (which might call an API route as discussed).

- **Template Library (`TemplateLibrary.jsx`):** A component that shows the list of templates. Likely uses a tree or list view. We can reuse a table or accordion UI from Tailwind UI components for listing templates grouped by category. Each item may have a preview button. If opened as a standalone page, selecting a template could open it in the Editor in a new document. If used within Editor (as a modal), selecting inserts content. We'll ensure the component is re-usable in both contexts.

- **Regulatory Search (`SearchPanel.jsx` or `SearchPage.jsx`):** If we want this as a global tool, it could be a page; but for convenience it might be nice as a slide-out panel in the Editor so that writers can search without leaving the doc. We'll design it as a panel with a search bar and results list (`SearchResultsList.jsx`, `SearchResultItem.jsx`). For each result, maybe an expand button to see more context. This component will call a search API route or use Supabase query directly (Supabase provides a JavaScript client that could do RPC calls or directly use full-text search if enabled). To integrate AI, the search component might also have a toggle for "semantic search" or an option like "Ask a question" which then calls OpenAI and displays an answer synthesized from the top results (with citations to the source docs). However, the base requirement is mainly finding precedent text and guidance, which the straightforward search covers.

- **Module Progress & Checklist (`ModuleProgress.jsx` and `ModuleDetail.jsx`):** The dashboard card shows a summary. We'll implement `ModuleProgress.jsx` to render the list of modules with completion bars. The data can come from an API like `/api/progress` that returns percentages and counts. When a user clicks a module, `ModuleDetail.jsx` opens (maybe as a modal or page) showing the checklist of documents. This component can allow direct actions like "Create Document" for a not-started section (which would jump into Editor with a template perhaps), or "View Document" if exists. It will fetch the list of required docs and their status from the backend (which can join the requirements with actual documents for that project). Possibly editing statuses (mark N/A or change status) could be allowed here with appropriate buttons.

- **Export Wizard (`ExportWizard.jsx`):** This component guides the user through export. It may have steps (we can use a multi-step form pattern): Step 1 – select region & sequence, Step 2 – confirm included modules/documents (with a checklist allowing them to deselect any optional docs), Step 3 – confirmation and generate. Or it could be a simpler form. After initiating export, it might show a spinner/progress bar. Once done, it shows a download link. We might also show a log of past exports in case they need to retrieve an older package (this could be another component listing previous export entries from a `Exports` table).

- **Navigation & Layout:** A top-level layout may include a header (with project name, user profile menu, etc.). Possibly, since TrialSage is multi-tenant (as indicated by "Acme CRO" in the screenshots), there might be an organization/project switcher. We would ensure the UI has a way to switch context if the user has multiple projects (this could be a dropdown in the header or part of the dashboard to choose a project, which then filters the documents and progress to that project).

All components will use Tailwind for styling, following a consistent design system (likely the one already used in existing UI). We will pay special attention to **responsiveness** (so users can at least view content on smaller screens, though heavy editing is likely desktop-only) and **accessibility** (proper ARIA labels, keyboard navigation especially for the editor and dialogs).

Backend Logic, API and Data Model

The backend is powered by **Supabase**, which gives us a Postgres database, row-level security, authentication, and storage out of the box. We will define the schema to support the features, and use Supabase's client library for CRUD operations from the frontend. Complex logic (like document conversion, XML generation, AI calls) might be implemented in a small Node.js/Express or Deno server (which could be hosted on Replit as well), or using Supabase Edge Functions (serverless functions in Supabase). Below is an outline of key tables and their relationships (the *data model*) and how the backend logic ties in:

Data Model (Postgres):

- **Organizations & Projects:** If multi-tenant, `Organizations` table (org_id, name) and `Projects` (project_id, org_id, name, type, region, etc.). For example, a project might be "IND Application for BTX-331" (as seen in screenshot) with region=FDA, type=IND. Modules and documents will link to a project. Users belong to orgs and have roles per project.

- **Users & Auth:** Supabase handles the Users table via its Auth. We can extend with a `UserProfiles` table for additional info/roles. Permissions can be enforced with policies (e.g., only users in the project's org can access its data).

- **Documents:** `Documents (doc_id, project_id, module, title, template_flag, status, current_version_id, tags)` – Stores one entry per document. `module` might be 1,2,3,4,5 corresponding to CTD modules. We might also have a field for the specific section or heading (like a code for which part of CTD it is, e.g., "2.5" or "1.3.4" as string) to align with the checklist. `template_flag` or a separate table indicates if this is a template. `status` could be an enum (Draft, In Review, Final). `current_version_id` points to the latest version in DocumentVersions.

- **DocumentVersions:** `DocumentVersions (version_id, doc_id, version_number, content_url, content_text, created_by, created_at, change_log)` – Each save creates a new version. `content_url` is a link to the file in storage (if we save the .docx or .pdf). `content_text` might store plain text or JSON of the content for quick search (optionally, could use for the search index instead of reading from file each time). `change_log` might be a short description of changes (the user can enter or auto-generated via comparing text). We may store only major versions or every save as a version depending on needs (could be a lot of versions, but Postgres can handle it and we can prune if needed).

- **Templates:** Could simply be Documents with `template_flag=true` and no project (or a special project like "Templates Repository"). Or a separate table if needed. Likely reuse Documents for simplicity. Templates might not need versioning like normal docs (or maybe they do for updates), but they can also use DocumentVersions.

- **ModuleRequirements:** `ModuleRequirements (id, region, submission_type, module, section_code, title, required)` – a table listing all possible documents. E.g., rows for "FDA, IND, Module 1, 1.3.4, Financial Certification, required=Y". This serves as the master checklist. We may pre-populate it with ICH Common Technical Document structure plus region-specific Module 1 items.

- **AuditLog:**
`AuditLog (log_id, project_id, user_id, action, object_type, object_id, timestamp, details)` – records events like "EDIT_DOCUMENT, doc_id=123 version 2", "EXPORT_PACKAGE, sequence=0001", etc.

- **Links/References:** If implementing cross-doc linking, possibly `DocumentLinks (source_doc_id, target_doc_id, description)` or perhaps links might just be stored as hyperlinks in content with a special scheme that can be resolved. A simpler approach: when generating output, scan for any internal links and replace with appropriate relative links.

- **Comments/Annotations:** If we add comments, a `Comments (comment_id, doc_id, author, text, range, created_at)` could store those, where `range` identifies the position in text.

- **Search Index:** If using full-text, we can add a tsvector column on DocumentVersions or Documents (for latest content). If using vectors, a table or extension to store embeddings, e.g., `DocumentEmbedding (doc_id, section_index, embedding vector(1536))` if using OpenAI embeddings of each section/paragraph.

These tables will be tied together with foreign keys (DocumentVersions -> Documents -> Projects -> Orgs, etc.). Supabase's row-level security will be configured so users can only access data for their org/project.

Backend Services & APIs:

- We will implement an **API layer** (could be part of the Replit app or separate) to handle complex operations. For instance:
- `POST /documents/:id/lock` – lock a doc for editing.
- `POST /documents/:id/unlock` – unlock after editing.

- `POST /documents/:id/version` – create a new version (save).
- `GET /documents/:id/version/:vid` – get a specific version content.
- `GET /search?query=...` – perform regulatory search (this endpoint will orchestrate the query to the database or vector index, and return results).
- `POST /export` – trigger an export (with payload specifying project and options).

Many simple CRUD interactions (like fetching document metadata, listing templates) can be done directly through Supabase's auto-generated APIs or client, but for the critical ones above we may write custom logic.

- **Document Conversion**: We might incorporate a service for handling .docx/PDF conversion as mentioned. Possibly a command-line tool or a library. On Replit, one could use a Node package for docx to pdf or route it to a cloud function. This would be invoked during export.

- **OpenAI Integration**: We will have server-side functions that call OpenAI. E.g., an endpoint `/ai/compliance-check` that receives text and returns identified issues or suggestions. Another `/ai/suggest` for more freeform suggestions. The prompt design will be crucial. (For example, for compliance we might prompt: *"You are a regulatory expert. The following text is part of an eCTD Module 2.6 Nonclinical Overview. Check it against ICH M4 nonclinical guidelines and list any missing elements or formatting issues."* and the AI returns a list of findings.) The backend formats this into a user-friendly response.

- **Notification/WebSockets**: If we want real-time collab (like others seeing updates or knowing when someone locks/unlocks a doc), we could use Supabase's real-time feature or a WebSocket. For now, perhaps not real collaborative editing (Google Docs style), but at least when a version is saved, others could get a notification to refresh. Supabase real-time can broadcast changes in the database (like new version row) to subscribed clients.

- **Performance and Scalability:** The backend design assumes a moderate number of documents (e.g., an IND might have a few hundred documents at most). Postgres can handle this. We will index important fields (like document by project, etc.). For large text search, using tsvector or vector indexes ensures quick lookup. The file storage will handle potentially large PDFs; Supabase allows files up to several MBs or more, which is fine for docs. We will enforce file size limits as per eCTD guidance (usually PDFs should be under 100 MB, etc.). Also, the XML generation will be mindful of sequence numbers if the submission is updated; perhaps out of scope for initial but worth noting for future (eCTD sequences).

In summary, the backend is the "brain" ensuring data integrity, security, and executing the heavy tasks (AI requests, file conversion, etc.), while the Postgres database serves as the Vault to persist everything reliably.

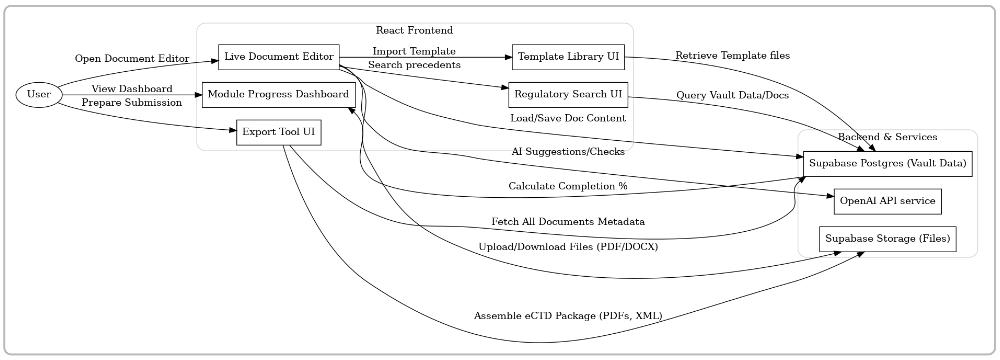


Figure 2: Conceptual architecture and workflow of the upgraded Co-Author Module. The user interacts with the React frontend, which comprises the live Document Editor, Template Library interface, Regulatory Search panel, Progress Dashboard, and Export tool. These frontend components communicate with backend services (Supabase Postgres acting as the "Vault" for data and Supabase Storage for files) and external APIs (OpenAI for AI assistance). The diagram illustrates key interactions: users load and save documents to the Vault, insert templates from the library, perform searches against Vault data, get AI suggestions for compliance, and finally the Export tool fetches all documents to assemble the eCTD submission package (generating PDFs and XML).

Module Workflow Example

To tie everything together, here's a typical workflow using the upgraded system:

1. **Project Setup & Template Selection:** The user (e.g., a regulatory writer) logs in and selects the project (submission) they are working on. On the dashboard, they see that Module 1 and 4 are only partially complete. They click "Module 4: Nonclinical" to view details and see a required document "2.6.4 Pharmacokinetics Written Summary" is not started. They click a "Create" action next to it. The system prompts them to choose a template — it opens the Template Library filtered to Module 2.6 (Nonclinical summaries) for FDA. The user selects the "Pharmacokinetics Written Summary – FDA template". A new document is created in the Vault and opened in the Editor with the template content loaded.

2. **Authoring with AI Assistance:** In the **Document Editor**, the user starts filling in sections of the document. The template has placeholders (like "[Study Results]"), which they replace with content. If they have existing text in a Word file, they import it (using an "Import Document" function to pull in a DOCX and convert to editable content). As they write, the AI Assistant highlights a sentence, suggesting "Consider clarifying the species for each study" or auto-formats a bulleted list per style. The user can at any time click "Check Compliance" — the system sends the current draft to the AI compliance service, which returns: *"Note: ICH guideline requires a summary of methods – not found yet."* The editor flags that so the author can add a Methods summary section. The assistant also automatically fixes a couple of capitalization errors and reminds the user to ensure all references are included in Module 5.

3. **Using Regulatory Search:** The author wants to see how a similar drug's pharmacokinetics summary was written. They open the Regulatory Search side panel and type a query or select a suggestion like "Search previous PK summaries". The search returns a result: "**Pharmacokinetics Summary – Compound X (NDA 2024)**" with a snippet about absorption and distribution. The user clicks it to view

more. They see a paragraph that fits well; with a click, they copy it as a starting point (perhaps marking it as quoted from another submission if needed). This saves them time, leveraging prior work.

4. **Saving and Versioning:** The user saves the document periodically. Each save creates a new version in the Vault. The user also writes a comment for the reviewer in the document. After finishing the first draft, they mark the document status as "In Review" and exit the editor, which automatically unlocks the doc. The Module Progress now shows Module 4's completion percentage bumped up, and that specific document is now "in progress" rather than not started.

5. **Collaborative Review:** A peer reviewer opens the same document (now read-only since it's in review status or locked for editing). They use a feature to add comments or suggestions. Alternatively, if minor edits are needed, they click "Edit" which locks the doc for them and maybe changes status back to Draft. They make changes (track changes is highlighting what they modified), then mark as "Final". Now Module 4 shows that document with a green check.

6. **Audit and Cross-links:** Meanwhile, an audit log entry was created for all these events (creation, edits, status change). If this PK summary document needs to reference a table or appendix that lives in Module 5, the writer inserts a cross-link (for example, a hyperlink to "Study ABC Plasma Levels in Module5/StudyReports.pdf"). The system ensures this link will be valid in the final PDF and adds a reference entry in a DocumentLinks table.

7. **Final Compilation:** Once all modules show 100% or acceptable completion, the user proceeds to **Export**. In the Export Tool, they choose "eCTD Package, region: FDA, sequence: 0001". They initiate the export. The backend gathers all documents from Vault (Module 1 admin forms, Module 2 summaries including the one just finished, Module 3 quality docs, etc.). It converts any remaining DOCX to PDF. It generates the XML backbone listing each file with correct titles. After a couple of minutes, the package is ready. The user downloads the ZIP, which contains folders for m1 through m5 and the index.xml, ready to be submitted via the FDA gateway. The user also opts to save a copy of this package in the Vault for record.

8. **Post-Submission:** If the submission needs updates or additional sequences, the system can be used to create a new sequence (increment sequence number, mark changes, etc., though that is beyond the initial scope). The key point is that all the content remains in the Vault for future reference, and the team can always go back to view exactly what was submitted (thanks to versioning).

Throughout this workflow, the combination of the live editor, AI assistance, vault versioning, templates, search, and automated packaging greatly streamlines what used to be a labor-intensive process. The user is guided at each step — from using the right template to verifying compliance and completeness, to assembling the output according to standards. This design thus transforms TrialSage's Co-Author Module into a comprehensive platform for regulatory document authoring and management, aligned with modern best practices in the industry.

References: The design draws on best-in-class concepts from regulatory tech. For example, **use of templates and metadata** is known to help create consistent, compliant documents with less effort [7], and connecting the authoring tool to a **document management system (Vault)** enables collaborative writing and reusability of content across projects [6]. By incorporating an AI co-author (similar in spirit to

Certara's *CoAuthor* tool [7]), we allow writers to focus on critical analysis while automating formatting and checks. Ultimately, this upgraded module will simplify and accelerate eCTD document preparation without sacrificing quality or compliance – providing an enterprise-grade solution on the Replit tech stack.

---

[1] **Real-Time Compliance Monitoring with AI: A New Era for Pharma ...**
https://www.freyrsolutions.com/blog/real-time-compliance-monitoring-with-ai-a-new-era-for-pharma-companies

[2] **How Artificial Intelligence is Transforming Regulatory Adherence**
https://www.tookitaki.com/compliance-hub/ai-in-compliance-how-artificial-intelligence-is-transforming-regulatory-adherence

[3] [4] [7] [8] [9] **Generative AI Tools for Regulatory Writing**
https://www.certara.com/blog/generative-ai-tools-for-regulatory-writing/

[5] **Working with Automated Linking & Link Evaluator | Vault Help**
https://regulatory.veevavault.help/en/lr/55400/

[6] [11] [12] [13] **How to Successfully Prepare for the Upcoming FDA eCTD Deadlines**
https://blog.montrium.com/blog/how-to-successfully-prepare-for-the-upcoming-fda-ectd-deadlines

[10] **Process Flow for Managing Regulatory Submissions - ClinSkill**
https://www.clinskill.com/docs/process-flow-for-managing-regulatory-submissions/