ChatGPT

TrialSage eCTD Co-Author System Implementation Plan

Overview: We will build the TrialSage eCTD Co-Author System – an intelligent regulatory document authoring platform – on Replit using a Node.js/Express backend, a React + Tailwind frontend, and Supabase for the database and auth [1] [2]. The system integrates Google Docs (for real-time co-authoring), Xerox DocuShare (as a regulatory DMS "vault"), and OpenAI GPT-4 (for AI assistance) to streamline creation of compliant eCTD submissions [3] [4]. Key features include a fully rendered eCTD document tree (Modules 1–5), an embedded Google Docs editor with OAuth2 login for collaborative editing, AI-powered writing assistance (Suggest, Validate, Draft), a controlled document lifecycle (Draft → Review → Final → Locked), role-based access control (Author/Reviewer/Approver/Admin), DocuShare vault integration for versioning, automated eCTD XML generation with PDF packaging, multi-region template support, a validation dashboard, and searchable submission history [5] [6]. The following steps break down the development process into a Replit-friendly playbook, complete with project structure, code examples, OAuth setup, and security considerations.

Step 1: Set Up the Replit Project and Tech Stack

Initialize the Replit environment: Start a new Replit project and configure it for a **Node.js + Express** backend and **React** frontend. On Replit, we can run a single container for both, serving the React app through the Node server. Organize the project into a monorepo structure with separate directories for backend and frontend code (e.g. `/backend` and `/frontend`). For example:

```
/project
├── backend/              # Node.js Express server
│   ├── package.json
│   ├── index.js          # Main server entry point
│   ├── routes/           # Express route modules
│   └── ... (controllers, utils, etc.)
├── frontend/             # React app (Tailwind CSS)
│   ├── package.json
│   ├── src/
│   │    ├── index.jsx    # React entry point
│   │    ├── App.jsx      # Main App component with routing
│   │    ├── components/  # Reusable UI components (Editor, Toolbar, Sidebar, etc.)
│   │    └── pages/       # Page components (Dashboard, EditorPage, Templates, Search, Export)
│   └── tailwind.config.js
└── replit.nix (optional) # Define Node, npm, etc., if needed
```

Install dependencies: In the backend, install Express and any needed libraries (e.g. `npm install express cors supabase pg jwt passport googleapis openai axios`). In the React frontend, install React, React Router, Tailwind CSS, and Supabase client (e.g. `npm install react-router-dom supabase-js`). Tailwind can be set up by installing `tailwindcss` and initializing a config. Configure Tailwind in `frontend/tailwind.config.js` and include the Tailwind directives in your main CSS file. Ensure the Replit project is configured to run both the build and server: for development, you can use `concurrently` to run the React dev server and Node server together, or for simplicity build the React app and serve static files via Express in production.

Replit Secrets: Use Replit's Secret Manager to securely store all credentials and config values. At minimum, define the following environment variables in Replit (these will be referenced in code):

- `SUPABASE_URL` and `SUPABASE_SERVICE_ROLE_KEY` (for database connection or using Supabase client).
- `GOOGLE_OAUTH_CLIENT_ID` and `GOOGLE_OAUTH_CLIENT_SECRET` (Google API credentials for OAuth2).
- `GOOGLE_OAUTH_REDIRECT_URL` (callback URI for Google OAuth – e.g. your Replit URL + `/auth/google/callback`).
- `DOCUSHARE_URL` (base URL for DocuShare API) and DocuShare credentials or API tokens (e.g. `DOCUSHARE_USERNAME`, `DOCUSHARE_PASSWORD` or a `DOCUSHARE_API_KEY` if provided).
- `OPENAI_API_KEY` (for GPT-4 access).
- Any other secrets (e.g. JWT signing secret if using custom auth).

On Replit, these secrets will be injected as environment variables. **Do not commit sensitive keys to the codebase.** By configuring them in Replit, they remain hidden and secure [7].

Supabase setup: Create a new Supabase project for this system. Note the project's **URL** and **service role API key** (use the service role key in the backend for full DB access). Enable Supabase Auth if desired (we will use it for storing users, though we might handle Google OAuth separately for Docs). In the Supabase dashboard, configure the database schema as described in the next step.

Step 2: Design the Database Schema (Supabase Postgres)

Using Supabase (which provides a hosted Postgres database), define tables to store users, documents, templates, and version history [6]. Below is a summary of key tables and their fields:

- **Organizations** – (org_id, name, etc.) Each tenant or company can be an org. This enables multi-tenant separation.
- **Projects** – (project_id, org_id, name, region, type, etc.) Represents an eCTD submission project (e.g. "IND Application for BTX-331"). Links to an organization and specifies region (FDA, EMA, PMDA) and submission type. Documents will belong to a project [8] [9].
- **Users / UserProfiles** – (user_id, name, email, role, org_id, google_token, docushare_token, etc.) Stores user info and roles. If using Supabase Auth, a base Users table is managed by Supabase; we can extend it with a UserProfiles table to store role and OAuth tokens [10] [11]. The **role** field can be an enum or text (Author, Reviewer, Approver, Admin) to enforce permissions. The Google OAuth

token (access + refresh) and DocuShare token for each user are stored here (encrypted or tokenized) so the backend can act on behalf of the user for those services [12].

- **Documents** – (doc_id, project_id, title, module, section_code, status, google_doc_id, docushare_id, current_version_id, created_by, updated_at, etc.) Each eCTD document/section authored in the system. Stores metadata: which project and module it belongs to, the CTD section code (e.g. "2.5" for Clinical Overview) to map to the CTD structure, the current workflow status (Draft/Review/Final/Locked), and links to external systems [13] [4]. For Google Docs integration, store the Google Doc ID here. For DocuShare vault, store the DocuShare document handle or ID. The **current_version_id** points to the latest entry in DocumentVersions.
- **DocumentVersions** – (version_id, doc_id, version_label, docushare_version, created_at, created_by, file_path) – Tracks each saved version of a document. When a document is moved along the workflow or explicitly versioned, we save a copy (e.g. PDF or DOCX) and log it here. The `docushare_version` might store the version number from DocuShare (e.g. DocuShare might assign Version 1, 2, etc.) or a link to the archived file in DocuShare. The `version_label` could be a friendly name (e.g. "Initial Draft", "Final v1.0").
- **Templates** – (template_id, region, section_code, title, google_doc_id, description) – Stores templates for various CTD sections and regions. Each template can be a Google Doc ID (if using Google Docs as template source) or stored file content. Templates help pre-populate new documents with proper structure. We differentiate templates by region if needed (since Module 1 differs by region) [14].
- **Comments/Reviews** (optional) – If implementing a review/comment system, a table to store comments on documents (comment_id, doc_id, author_id, text, resolved_flag, etc.).
- **SubmissionExports** (optional) – To log compiled submissions (export_id, project_id, sequence_no, generated_at, generated_by, file_path). This can record each time a submission ZIP is generated, storing the sequence number and location of the package (e.g. in Supabase Storage or Replit file storage).

Define the schema either via SQL migration scripts or the Supabase GUI. For example, an SQL snippet for **Documents** table:

```
CREATE TABLE Documents (
  doc_id            SERIAL PRIMARY KEY,
  project_id        INTEGER REFERENCES Projects(project_id),
  title             TEXT NOT NULL,
  module            INTEGER NOT NULL,        -- e.g. 1, 2, 3, 4, or 5
  section_code      TEXT,                    -- e.g. '2.5.1'
  status            TEXT NOT NULL,           -- 'Draft', 'In Review', 'Final', 'Locked'
  google_doc_id     TEXT,
  docushare_id      TEXT,
  current_version_id INTEGER REFERENCES DocumentVersions(version_id),
  created_by        INTEGER REFERENCES Users(user_id),
  updated_at        TIMESTAMP DEFAULT now()
);
```

Ensure **relationships**: Each Document links to a Project (which links to an Org). Use foreign keys so that, for example, deleting a project can cascade to its documents (or better, simply mark projects inactive for audit

trail). The schema will enforce basic integrity, while application logic will enforce eCTD-specific rules (like one document per required section, etc.).

Row-Level Security (RLS): Configure RLS policies in Supabase to isolate data by organization for multi-tenant security [9] [15]. For instance, on the Documents table, add a policy so that a user can select documents only if `document.project_id` belongs to an organization that the user is a member of. Similar policies apply to other tables. This ensures that even if a malicious request is made, a user cannot access another org's data.

Seed data: Optionally, pre-populate the **Templates** table with some standard eCTD templates. For example, you might load Module 2 summaries templates or Module 3 quality document templates, possibly inspired by Celegence's Dosscriber content (which emphasizes consistent styles and auto-generated TOCs for each region's requirements [14]). Also seed a default admin user (if not using Supabase Auth) and some organization/project entries to test the flow.

Step 3: Implement OAuth2 Authentication (Google and DocuShare)

Google OAuth2 Setup: We will use Google's OAuth2 to allow users to log in with Google and grant the app access to Google Docs/Drive for editing and export. In the Google Cloud Console, create an OAuth **Web Application** credential. Set the **authorized redirect URI** to your Replit app URL + `/auth/google/callback` (e.g. `https://TrialSage--yourusername.repl.co/auth/google/callback`). Enable the Google Drive and Google Docs APIs for your project. For scopes, request at minimum: - `https://www.googleapis.com/auth/drive.file` (read/write files created by the app) or `.../drive` for broader access if needed, - `https://www.googleapis.com/auth/documents` (Google Docs API), - plus `openid email profile` if using it for sign-in identity.

Store the Google OAuth **Client ID** and **Client Secret** in Replit secrets (`GOOGLE_OAUTH_CLIENT_ID`, etc.). In Express, set up the Google OAuth flow:

```
// backend/routes/auth.js (example)
const { google } = require('googleapis');
const oauth2Client = new google.auth.OAuth2(
    process.env.GOOGLE_OAUTH_CLIENT_ID,
    process.env.GOOGLE_OAUTH_CLIENT_SECRET,
    process.env.GOOGLE_OAUTH_REDIRECT_URL
);

// Step 1: Initiate OAuth flow
app.get('/auth/google', (req, res) => {
  const scopes = [
    'https://www.googleapis.com/auth/drive.file',
    'https://www.googleapis.com/auth/documents',
    'profile', 'email'
  ];
  const authUrl = oauth2Client.generateAuthUrl({ scope: scopes, access_type: 'offline', prompt: '
  res.redirect(authUrl);
```

```
});

// Step 2: OAuth callback endpoint
app.get('/auth/google/callback', async (req, res) => {
  const { code } = req.query;
  const { tokens } = await oauth2Client.getToken(code);
  // tokens contains access_token, refresh_token, etc.
  oauth2Client.setCredentials(tokens);
  const oauth2 = google.oauth2({ version: 'v2', auth: oauth2Client });
  const userInfo = await oauth2.userinfo.get();  // get user's Google profile
  const email = userInfo.data.email;
  // TODO: Create or fetch user in our database by this email
  // Save tokens (encrypted) to the user record for future API calls
  // Establish application session (e.g., JWT or cookie)
  res.redirect('/');  // redirect to frontend app
});
```

In the callback, once we get the Google tokens, link them to our app's user. If using Supabase Auth, you might create a Supabase user with the Google email at first login. Or store a session JWT (signed with a secret) that the frontend will use for subsequent API calls. The Google **refresh_token** is crucial – store it securely (so our server can refresh the Google access token when needed without user re-login).

DocuShare Authentication: Xerox DocuShare's cloud API may support either an API key or an OAuth-like flow. We will integrate DocuShare as our secure document repository (the "Vault"). If DocuShare provides OAuth or token-based auth, set up similarly: get client credentials from the DocuShare admin. Otherwise, we handle user credentials directly. For simplicity, let's assume DocuShare API allows a service account or user login via REST.

In Express, create an endpoint to handle DocuShare login or token fetch. For example:

```
app.post('/auth/docushare', async (req, res) => {
  const { username, password } = req.body;
  // Example using basic auth to get a session token from DocuShare API
  try {
    const dsResponse = await axios.post(`${process.env.DOCUSHARE_URL}/api/v1/auth/login`,
       {},
       { auth: { username, password } });
    const dsToken = dsResponse.data.token;
    // Save dsToken to user's profile in DB (associated with userId)
    // (In DocuShare, this token might be a session cookie or API key to use on subsequent reques
    res.json({ success: true });
  } catch (err) {
    res.status(401).json({ error: 'DocuShare login failed' });
  }
});
```

The exact API path will depend on DocuShare's API (this is an illustrative example). Some DocuShare deployments might require using their SOAP services or a specific SDK, but many modern DMS have REST endpoints for authentication and file upload. If DocuShare supports OAuth2, the flow would be similar to Google's (redirect to DocuShare's auth page, etc.). In any case, store whatever token or credentials are needed in the database tied to the user (or a service account) [12] [16]. This ensures the backend can perform DocuShare operations (upload, version, retrieve files) on behalf of the user/project.

Session Management: After a user authenticates, establish a session for your app. If using Supabase Auth, the frontend can use Supabase's JWT. Alternatively, issue a JWT in Express (signed with a secret from Replit env) containing the user's ID and role. Send this JWT as a cookie or in the response, and have the React app store it (e.g. in `localStorage` or an HttpOnly cookie). This token will be included in subsequent API calls (e.g. via `Authorization: Bearer ...` header) to authorize the user. All protected API routes should verify this JWT and attach the user identity to the request (you can use middleware for JWT verification). From now on, the user is logged into the TrialSage app and we have their Google/DocuShare tokens stored for external integrations.

Step 4: Develop Core Backend APIs (Documents, Templates, Workflow)

Implement the Express routes that the frontend will call for all main actions. We will create a dedicated router module for each domain (e.g. `documents.js`, `templates.js`, `workflow.js`, `ai.js`). All routes will enforce that a valid session token is present (and optionally check roles/permissions per action). Below are the key endpoints and their logic [17] [18]:

- **GET** `/documents` – List documents the user has access to (likely filter by project). Returns metadata for each doc: ID, title, module, status, last modified, etc. [19]. Support query parameters to filter by module or status. In the handler, query the Documents table for the current project (ensuring via RLS or where clause that the user's org/project only is returned). Example:

```
app.get('/documents', async (req, res) => {
  const userId = req.user.id;
  const projectId = req.query.project;  // assume frontend provides current project
  // Fetch docs for this project from DB
  const docs = await db.query(
    `SELECT doc_id, title, module, section_code, status, updated_at
       FROM Documents
       WHERE project_id=$1`, [projectId]);
  res.json(docs.rows);
});
```

- **POST** `/documents` – Create a new document. The frontend will supply the title, module, section (or template selection) and project. The server will:

- Call the Google Drive API to create a new Google Doc. This can be done by copying a template document if a template_id is provided (Google's API `drive.files.copy` allows copying an existing doc to preserve formatting). For example:

```
const drive = google.drive({ version: 'v3', auth: oauth2Client });
let fileMetadata = { name: req.body.title, mimeType: 'application/vnd.google-apps.document'
if (req.body.templateDocId) {
  // copy from an existing template in Drive
  const newDoc = await drive.files.copy({
    fileId: req.body.templateDocId,
    resource: { name: req.body.title }
  });
  googleDocId = newDoc.data.id;
} else {
  // create blank document
  const newDoc = await drive.files.create({ resource: fileMetadata });
  googleDocId = newDoc.data.id;
}
// Optionally, use Docs API to insert some initial content or placeholders.
```

- Call DocuShare API to create a placeholder record for this document. This might involve creating a document object in a specific DocuShare collection or folder for the project. For example, a POST to something like `/docs/create` on DocuShare with metadata (title, project) to reserve an ID. The response would include a DocuShare document ID or handle (e.g. "DOC-12345"). Save that.

- Insert a new row in the **Documents** table with all pertinent info: title, module, section, project, initial status = "Draft", the `google_doc_id` from step 1, and `docushare_id` from step 2 [20].
- Return the new document record (including its generated `doc_id` and perhaps an embed URL or the Google Doc ID for the frontend to load).

This ensures a new Google Doc is ready for editing and a corresponding entry exists in the DocuShare vault (even if empty initially) for version tracking.

- **GET** `/documents/:id` – Fetch detailed info for a specific document. This will retrieve all metadata and possibly a dynamically generated **Google Docs embed URL**. The frontend needs a URL to load the Google Doc in an iframe. We can construct it as `https://docs.google.com/document/d/[google_doc_id]/edit`. If the user is already authenticated with Google (via our OAuth), and the document is either owned by or shared with the user, it will load in edit mode. We might include an OAuth access token in a header rather than URL (to avoid exposing it), but since the iframe is directly to Google, the user's Google session or cookie will handle auth. The GET could return something like:

```
{
  "doc_id": 12,
  "title": "Clinical Overview",
  "module": 2,
  "section_code": "2.5",
  "status": "Draft",
  "google_doc_id": "1AbCdEfGhIjKlMnOpQRsT",
```

```
  "embed_url": "https://docs.google.com/document/d/1AbCdEfGhIjKlMnOpQRsT/edit"
}
```

The embed_url is constructed on the fly [21] [22]. (If needed, ensure via Google API that the user has permission to access that doc – e.g. share the doc with the user's Google account if the doc is owned by a service account.)

- **POST** `/documents/:id/save` – Save a version of the document. Given Google Docs auto-saves, this may not be called frequently, but when a user explicitly wants to capture a version or when transitioning state, we use this. The server will use Google Drive API to **export** the Google Doc content (to PDF and/or DOCX) and then upload that file to DocuShare:
- Use Drive API's export endpoint: `GET https://www.googleapis.com/drive/v3/files/{googleDocId}/export?mimeType=application/pdf` to get PDF content (or use the Node client library to fetch and pipe the file).
- Save the PDF to a temporary location (e.g. `/mnt/data/documents/[docId]-vX.pdf`).
- Call DocuShare API to upload this file as a **new version** of the document. DocuShare likely has an endpoint for updating a document's content (e.g. PUT to `/docs/{docushare_id}` with the file). The DocuShare system will then assign a version number (like incrementing Version-1 to Version-2).
- Insert a new row in **DocumentVersions** table with details of this version (and update Documents.current_version_id).

This endpoint returns success or the new version info. It might not be needed if we integrate save with state changes, but it's useful for manual versioning.

- **POST** `/documents/:id/state` – Change the state (lifecycle stage) of a document [23]. This enforces the workflow transitions Draft → Review → Final → Locked. Include in the request the target state (and perhaps who is approving it, etc.). The handler will check the current status in DB and ensure the move is allowed (e.g. cannot go backwards from Final to Draft, unless admin override). If moving to **Review** or **Final**:
- If moving to **In Review**: We might update the doc status and perhaps trigger an email or notification to assigned reviewers (notification system optional).
- If moving to **Final**: This is critical – we should capture the final content. This is a point to call `/documents/:id/save` internally to freeze a version. So on Final, export the Google Doc to PDF and push to DocuShare (if not already done). Mark that version as final. We might also set the Google Doc to read-only mode (for instance, revoke edit permissions for authors or lock it in DocuShare) since it's final. Then update status = Final.
- If moving to **Locked**: Once a submission is compiled or the document is officially approved, Locked means it cannot be edited further in the system. The Google Doc might be completely locked down (could remove all editors or even delete it if archival is solely in DocuShare). In the app, we'll simply restrict editing on Locked docs (only viewing the archived PDF).

Implement the state change with appropriate side-effects: create a DocumentVersions entry if needed, update DB, and enforce any permission changes. Respond with the updated document status.

- **GET** `/templates` – List available templates. This can return template entries filtered by region or module (e.g. query params `?region=EMA&module=2`). The result is used in a Template Library UI for users to choose a template for new docs. It should include template metadata and maybe an ID

or link if the user wants to view the template. If templates are stored as Google Docs, you might include a read-only link or the Google Doc ID.

- **POST** `/templates` – (Admin only) Create or update a template. Allow an admin to upload a new template file or link a Google Doc as a template. This would insert/update the Templates table. If uploading via the app, you could use Supabase Storage to store the file or directly create a Google Doc as the template repository.

- **GET** `/search` – Search across documents (and possibly templates or prior submissions). Accept a query string and perform a full-text search on relevant content. This can be a simple SQL `to_tsvector` search on document text stored in the database, or an advanced semantic search using OpenAI embeddings. Initially, implement basic search: e.g., search by document titles or keywords in a "keywords" field. We might expand this later (see Step 10). Return a list of matching documents or past submissions. This helps users find existing text to reuse or reference (a feature to inject prior content) [24].

Make sure to use Express **middleware** to protect these routes. For example, add a JWT verification middleware that also checks user role for certain routes (only Admin can access template creation, only Approver can finalize documents, etc.). Express can also use CORS middleware to allow the frontend origin.

Throughout these backend functions, integrate with the external APIs as needed, using the tokens we stored: - Use the Google OAuth token from the user's session or DB to auth Google API calls (instantiate a google API client each time or store the oauth2Client with credentials). - Use the DocuShare token for DocuShare API calls (e.g. include in headers).

By structuring the backend as above, we cover creating, editing, saving, transitioning, listing, searching, and template management – effectively all non-AI aspects of document co-authoring.

Step 5: Integrate GPT-4 AI Assistant Endpoints (Suggest, Draft, Validate)

One defining feature is the AI Co-Author assistant powered by GPT-4 [25]. We will implement backend endpoints that the frontend can call to get AI-generated content suggestions. There are three main AI functions: **Suggest** (contextual suggestions while writing), **Draft Section** (generate initial content for a blank section), and **Validate** (compliance or quality checking of a draft). We will use the OpenAI API (GPT-4 model) with carefully engineered prompts for each task [26] [27].

Set your OpenAI API key in Replit secrets (`OPENAI_API_KEY`). Install OpenAI's library (`npm install openai`) or use direct HTTP calls. We will create an Express router for AI, e.g. `ai.js`:

- **POST** `/ai/suggest` – Provides a suggestion or continuation. The request should include either the document ID or a snippet of text from the editor. For example, the frontend might send the last paragraph or last few sentences the user wrote, and ask for a suggestion to continue or improve it [28]. The backend will then construct a prompt for GPT-4. **Prompt engineering** example for Suggestion:

```
// Pseudocode inside /ai/suggest route:
const { docId, text, instruction } = req.body;
// Fetch any needed context, e.g., the document title or section, to guide AI.
const prompt =
  `You are an expert regulatory medical writer assisting with an eCTD submission document.
  The user is writing "${documentTitle}" (CTD section ${sectionCode}).
  The current content is:\n"""${text}"""\n
  Please suggest the next sentence or an improvement, maintaining a formal tone and ensuring regu
const response = await openai.createCompletion({
  model: 'gpt-4',
  prompt: prompt,
  max_tokens: 100,
  temperature: 0.7,
  // other params
});
res.json({ suggestion: response.data.choices[0].text.trim() });
```

The prompt establishes the AI's **role** (expert regulatory writer) and provides context (section title, existing text) so GPT-4 produces a relevant suggestion [29] [30]. For example, if the user last wrote about a study result, the AI might suggest the next logical point. The backend simply relays the AI's suggestion back to the frontend.

- **POST** `/ai/draft` – Draft a section. This is used when a section is empty or the user wants AI to create a first draft. The request might include the section identifier and some parameters (e.g. a brief outline or just the section name and any notes). The backend will craft a prompt instructing GPT-4 to generate a thorough draft for that section. **Prompt example for Draft:**

```
const { sectionCode, sectionTitle, context } = req.body;
const prompt =
  `You are an expert regulatory writer with deep knowledge of ICH eCTD guidelines.
  Draft the content for the section "${sectionTitle}" (CTD ${sectionCode}) of a regulatory submis
  ${ context ? "Here are relevant details to include:\n" + context : "" }
  The draft should be comprehensive, following the common technical document format, and written
// Call OpenAI with a high max_tokens limit to get a multi-paragraph draft
```

Here we set the stage for GPT-4 to act as a knowledgeable writer and produce a full section draft. If `context` is provided, we incorporate it (e.g. bullet points the user gave, or summaries from other sections). If the section is something like a **Clinical Overview**, we might provide context from Module 5 study reports. This approach mirrors how Certara's CoAuthor uses a specialized GPT model with an eCTD template library to streamline writing [31]. By giving GPT a template of what the section should cover, we leverage AI to dramatically speed up initial drafting (Certara reported >20% reduction in drafting time with such technology [32]).

- **POST** `/ai/validate` – Validate a document's content for completeness and compliance. This will send the full text (or a large chunk) of a section or document to GPT-4 and ask it to act as a QA

checker for regulatory standards [33] [34]. Since GPT-4 can handle long prompts, we can include a summary of relevant guidelines. **Prompt example for Validate:**

```
const { docId, text } = req.body;
// Possibly determine which section or module to know what guidelines apply
const prompt =
  `You are an expert regulatory reviewer with deep knowledge of FDA and ICH eCTD guidelines.
  You are reviewing the following content for completeness and compliance with those guidelines.
  List any issues, such as missing required information, sections that seem incomplete, or deviat
  Content to review:
  """${text}"""`;
const response = await openai.createCompletion({ ... });
```

The AI's response might be a list like: "1. Missing summary of XYZ study. 2. No mention of safety data. 3. The section headings do not follow ICH structure..." etc. We then return this list to the frontend. This helps the user ensure the document meets requirements (acting as an automated reviewer). We align this with known regulatory checklists – effectively asking GPT-4 to emulate a QA specialist [34].

All AI calls should include our OpenAI API key on the request. Make sure to handle errors (e.g., API timeouts or content length issues). Also, consider limiting the frequency or length of calls to manage costs and latency. The prompts may evolve with tuning, but the above examples give a starting point for each feature. We also incorporate learnings from existing solutions: for instance, instruct GPT-4 with a role like *"You are an expert in regulatory writing"* to yield more precise outputs [35].

Note on prompt tuning: We might maintain a set of prompt templates for each AI feature and each module type. For example, a Module 3 (Quality) section might require a different tone or checklist than a Module 2 summary. We can adapt the prompt with module-specific hints (e.g., "This is a quality/CMC document, ensure to include manufacturing details...") [27]. This ensures the AI output is context-appropriate.

Step 6: Implement Document Lifecycle Workflow (Draft → Review → Final → Locked)

Implementing the document **lifecycle** involves both backend enforcement and frontend controls. The statuses (Draft, In Review, Final, Locked) represent stages of document maturation and should strictly flow forward [36].

Backend Enforcement: In the `/documents/:id/state` endpoint (from Step 4), we enforce allowed transitions. Define a simple state machine logic:

```
const validTransitions = {
  'Draft': ['In Review'],
  'In Review': ['Final', 'Draft'],
  'Final': ['Locked'],
```

```
  'Locked': []
};
```

As a business rule, maybe allow going from Review back to Draft if revisions are needed, but not from Final back to Draft (unless admin override). Attempting an invalid transition returns an error. This ensures, for example, only a Reviewer or Approver can move something to Final, and once Locked, no further edits occur [37].

Lifecycle Actions: - When a document is moved to **In Review**, the system might set the Google Doc to "Suggestion Mode" or view-only for Authors, and allow Reviewers to add comments or suggestions. While we cannot programmatically toggle Google Doc modes easily via API, we can enforce in UI: if status=Review, open the Google Doc in suggestion mode (Google supports a query param `?mode=suggest` on docs URLs for suggesting). Alternatively, instruct users to use suggesting in Google. We can also log the time it entered review and who initiated it. - On **Final**, as described, export and archive the content. We should ensure all edits are captured. Possibly disable the Google Doc's editing for everyone (e.g., via Drive API, remove others' edit permissions or make it read-only). Also, mark the DocuShare record as final (maybe set a property or simply rely on versioning). - **Locked** means the document is part of a submitted sequence (or otherwise frozen). At this stage, the backend might even remove the Google Doc (to enforce no changes) or keep it for reference but certainly not allow further editing through the app. In DocuShare, everything is already saved. In our DB, status Locked can trigger the UI to not even load the Google iframe (instead perhaps show a PDF view of the final content).

Frontend Controls: The React app should reflect the state: - If a doc is Draft, show editing enabled. - If In Review, maybe show a banner "In Review – read-only for authors" and allow only comments or suggestions. - If Final, show "Final – awaiting Lock" or if submission is compiled, etc., and provide a read-only view. - If Locked, indicate it's locked and just show the content (download PDF link).

For moving between states, only certain roles can see the buttons: e.g. an Author can click "Submit for Review" (to go Draft -> Review), a Reviewer or Approver can click "Approve Final" (Review -> Final), and maybe only Admin can lock (Final -> Locked) or it happens automatically during export.

We also log these transitions for audit trail. Each state change can create an entry in a **DocumentEvents** or audit log table (user X changed document Y to Final on date Z).

Implementing this workflow ensures document status is clear and editing privileges are controlled, a critical compliance need for regulatory writing [36].

Step 7: Implement Role-Based Access Control (RBAC)

Enforce role-based permissions throughout the system so that Authors, Reviewers, Approvers (and Admins) have appropriate access:

- **Author:** Can create and edit documents (in Draft), request review, and view content. Cannot finalize or approve.
- **Reviewer:** Can edit (or at least comment on) documents in Review, and recommend changes. May be allowed to move document from Review back to Draft (if revisions needed).

- **Approver:** Can mark documents as Final. Approvers (or Admin) can Lock documents (or that might be done automatically on submission).
- **Admin:** Can do all the above plus manage templates, manage users, and perhaps see all projects.

These roles are stored in the user profile (and possibly per project). We must check roles in sensitive API endpoints: - In `/documents/:id/state`, if trying to go to Final or Locked, ensure `req.user.role` is Approver or Admin. - In `/templates` management, ensure Admin role. - In `/ai/*`, maybe all roles can use AI suggestions (as long as logged in), but possibly track usage.

We can implement a simple middleware, e.g.:

```
function requireRole(role) {
  return (req, res, next) => {
    const userRole = req.user.role;
    const roles = Array.isArray(role) ? role : [role];
    if (!roles.includes(userRole)) {
      return res.status(403).json({ error: 'Forbidden' });
    }
    next();
  };
}

// Example: only Admin can create template
app.post('/templates', requireRole('Admin'), (req, res) => { ... });
```

On the frontend, use the user's role (stored in context or derived from token) to conditionally render UI elements: - Don't show "Finalize" button to Authors. - Show Template Library management only to Admin. - Possibly have a "Review Dashboard" where reviewers see docs awaiting their review – only accessible to Reviewer/Approver.

Supabase Auth can also enforce some of this if we set up policy groups, but since we manage roles in our app, the above approach suffices. The **Users** table's role field (or a join table UserProjectRole) should be set when inviting/adding users to a project. For example, an Admin user can invite a colleague as a Reviewer to the project, which our backend would record.

With RBAC in place, we prevent unauthorized actions. For instance, a user without proper role cannot manually call the finalize API and mark a document Final – the check will reject it. This maintains the integrity of the review/approval process, satisfying regulatory expectations for controlled document authoring.

Step 8: Integrate DocuShare Vault DMS (Versioning & Archival)

Vault integration ensures that every official document version is stored in a compliant Document Management System (Xerox DocuShare) [4]. We partially integrated DocuShare in earlier steps (creating placeholders and uploading versions). Now, finalize the integration:

- **Storing Versions in DocuShare:** For each significant event (e.g. a document moves to Final or a user clicks Save Version), upload the latest content to DocuShare. Use the DocuShare API to either **create** a new document (for first-time Final) or **check in** a new version (for subsequent updates). DocuShare typically assigns each document a unique ID/handle on creation and manages versioning internally (like Version 1, 2, etc.). We store that base ID in `Documents.docushare_id`. On each new version, the DocuShare response might indicate the new version number or a version-specific ID; record that in DocumentVersions.

- **Retrieving from DocuShare:** In case we want to allow viewing past versions or ensure the final PDF is what's in DocuShare, we can fetch from DocuShare when needed. For example, a **VersionHistoryModal** in the UI can call an API `/documents/:id/versions` which pulls version metadata from our DB and possibly fetches file links from DocuShare. If DocuShare provides direct download links for versions, the backend can relay those (potentially requiring a DocuShare token).

- **DocuShare API client:** Implement a small utility module for DocuShare operations (if no official SDK). For instance:

```
// pseudocode for uploading a file to DocuShare
async function uploadToDocuShare(docushareId, filePath, token) {
  const fileStream = fs.createReadStream(filePath);
  await axios.put(`${DOCUSHARE_URL}/api/v1/documents/${docushareId}`, fileStream, {
    headers: { 'Authorization': `Bearer ${token}`, 'Content-Type': 'application/pdf' }
  });
}
```

  The actual endpoint and auth will depend on DocuShare's API spec (adjust accordingly).

- **Document finalization in DocuShare:** After uploading the final version, you might want to "lock" or finalize the doc in DocuShare. Some DMS allow setting a document status to approved or locked, preventing further changes. If such an option exists, use it when status → Locked.

- **Synchronization considerations:** In our design, the Google Doc is the live editing platform and DocuShare is the system of record. We should reconcile differences carefully:

- If users edit content in Google Docs, those changes are saved in Google until we explicitly push to DocuShare. Make sure final changes make it over.
- If needed, schedule periodic background syncs (e.g. every X hours save a version to DocuShare) to avoid large deltas, but that may be overkill.

- The one-click **Export to eCTD** (coming next) will rely on DocuShare-stored PDFs or Google exports. We can either use the fresh Google export at compile time or use the DocuShare version (assuming the latest is there). Using DocuShare as the source of truth is safer after Final.

By integrating DocuShare, we ensure compliance: **audit trails** and **version history** are maintained outside the editing environment. This satisfies regulatory requirements for document control (DocuShare holds the "single source of truth" for each version [4] [38]). It also means if our app goes down, documents still reside in the enterprise vault.

Finally, test the DocuShare integration thoroughly with a sandbox environment: create a doc, upload versions, retrieve versions, etc., to confirm the API endpoints and our usage align. Once working, all document archival will happen seamlessly as users progress their documents.

Step 9: Implement eCTD Export (PDF Conversion, XML Mapping, ZIP Packaging)

One of the ultimate goals is **one-click export** of an eCTD submission package (the set of documents) into a valid eCTD ZIP, including the XML backbone [36]. We'll implement a backend process triggered by an API (e.g. `POST /submissions/export`) that gathers all final documents for a project and produces the submission structure.

Export Trigger: The frontend might provide an **Export Wizard** where the user selects options (region, sequence number, which modules to include) and then clicks "Generate eCTD Package". When confirmed, it calls our backend `/submissions/export` endpoint with the project ID and desired sequence number, etc. [39].

Gather Documents: The backend will fetch all Documents for the project where status = Final (or Locked) – these are the ones to include. For each document, retrieve the latest PDF: - If we've been storing PDFs in DocuShare for finals, we can fetch from DocuShare (this ensures we use the archived copy). - Alternatively, use Google Drive API to export the doc again to PDF on-the-fly (provided the content hasn't changed since finalization, it should be the same).

File Structure: eCTD submissions have a specific folder structure: typically, a root folder with an **index.xml**, and subfolders for each module (e.g. `m1`, `m2`, ..., `m5`). Within those, further subfolders per CTD section, etc., following ICH guidelines (for instance, Module 2 might have `m2/25-clinical-overview/` containing that document PDF). The structure can be complex, but for a simple approach: - Create a temp directory `./export/[sequence]/` (sequence number like 0000, 0001, etc., depending on whether this is the first or subsequent). - Under it, create subdirectories `m1`, `m2`, `m3`, `m4`, `m5`. - For each document, determine its module and section_code. Place the PDF into the appropriate directory. We might use the section code to name the file or folder. For example, a document with section_code "2.5" (Clinical Overview) could be placed at `m2/25-clinical-overview/25-clinical-overview.pdf`. Use consistent naming (no spaces, perhaps use section number and a slug of title). We can also create a simple mapping: e.g., replace dots with nothing or underscore in section code.

Generate index.xml: The index.xml is the backbone that lists all documents (leaf elements) and their metadata (operation, checksum, etc.). We will programmatically generate it. We know: - Each document's file

path and name, - The section (we can derive the CTD heading from section_code or store a mapping in a config).

For each document, create an XML `<leaf>` entry. Example (pseudo-XML for a Module 2 document):

```
<leaf ID="leaf0001" xlink:href="m2/25-clinical-overview/25-clinical-overview.pdf" operation="new"
  <title>2.5 Clinical Overview</title>
</leaf>
```

If this is sequence 0000 (the first), operation is “new” for all. If subsequent, some might be “replace” or “delete” but that’s beyond initial scope. We can assume initial submission for now. We should calculate the MD5 checksum of each PDF and include it (this is required in eCTD backbone). Use Node’s crypto library to compute MD5 of each file.

Also consider region-specific requirements: - For FDA, there’s usually a US regional.xml for Module 1 (admin info) and the backbone index.xml for Modules 2-5. But it might suffice to include everything in one index.xml referencing Module 1 docs as well, depending on spec. We might generate a separate `us-regional.xml` if needed (for simplicity, we can skip that detail or create a dummy if FDA requires it). - For EMA, the envelope is slightly different but still an index.xml. We will assume ICH format for now.

Use an XML builder library or string templates to construct the XML. For reliability, you might have an XML template file with placeholders. But given this is dynamic, a snippet approach is fine. For example, in Node:

```
let xmlContent = '<?xml version="1.0" encoding="UTF-8"?><ectd:ectd xmlns:ectd="http://ich.org/ect
xmlContent += '<m1-administrative-information>';
// ... add Module 1 leaves
xmlContent += '</m1-administrative-information><m2-common-technical-document>';
// ... Module 2 leaves, etc.
xmlContent += '</ectd:ectd>';
fs.writeFileSync(`./export/${sequence}/index.xml`, xmlContent);
```

(This is a simplified representation; real eCTD XML has more nested structure. For a demo, focusing on leaves with correct file paths is key.)

Packaging: Once all PDFs are in place and XML files generated, create a ZIP archive of the `./export/[sequence]` folder. You can use Node’s built-in `zlib` or a package like `archiver`. For example:

```
const archiver = require('archiver');
const output = fs.createWriteStream(`./export/Submission_${sequence}.zip`);
const archive = archiver('zip');
archive.pipe(output);
archive.directory(`./export/${sequence}/`, false);
await archive.finalize();
```

This produces `Submission_<seq>.zip` containing the eCTD folder structure. Mark it with the sequence or project name for clarity.

After zipping, respond to the API call with a download link or path. Since the app is on Replit, one approach is to upload this ZIP to **Supabase Storage** (which can serve as a temporary file store for downloads). Or you can base64 encode and send (not ideal for large files). A simpler way: have the frontend call another GET like `/submissions/download?file=Submission_0000.zip` which then serves the file from Replit's filesystem (if accessible). Replit might allow serving static files; if not, storing in Supabase or S3 and returning a URL is fine.

Export UI feedback: The Export Wizard (frontend) should show progress or at least indicate generation. The backend could either do it synchronously (potentially long if many files) or kick off a background job. On Replit, background processing is limited, but since this is interactive and likely not too large (a dozen PDFs), doing it on the fly is fine. Stream the result or notify when ready. The UI can then present a **Download Submission Package** link.

At this stage, the user can take the ZIP and upload to the FDA or other agency's submission portal, confident that the structure is correct. Our system automates what was previously a manual packaging effort, achieving the "one-click export to a valid eCTD package" goal [36].

Step 10: Build the React Frontend – User Interface and Components

With backend and core logic in place, implement the frontend in **React + Tailwind CSS** to deliver a rich user experience. The UI should resemble a modern eCTD authoring tool, with a dashboard, an editor page with sidebars, and supporting pages for templates, search, and export. We will create React components for each major part of the app, ensuring the design is modular and state is managed (you can use React Context or state management libraries for global state like current project, user info, etc.). The styling will use Tailwind utility classes for consistency (e.g., applying Veeva Vault-like styling for enterprise feel [40]).

1. Dashboard Page (Home) – This is the landing after login. It should summarize the project and allow navigation to other features [41]. Key elements: - **Project Selector:** If the user has multiple projects (multi-tenant or multiple submissions), allow switching context. This could be a dropdown in a navbar (e.g. showing "Acme CRO – IND BTX-331" as in our example). Selecting a project updates the context (possibly stored in React state or context provider). - **Recent Documents:** A list of documents the user recently worked on (call GET `/documents?project=X` with perhaps a limit/sort). Display the title, module, and status, with a link or button to open the editor [42]. - **Module Progress Overview:** A summary of completion status per module [43]. This can be displayed as a list or set of progress bars. E.g., "Module 1: 80% complete, Module 2: 60%" based on how many required docs are Final. The backend can compute these, or the frontend can compute after fetching all docs (e.g., if we know total required sections vs final count). Show each module with a percentage or fraction, possibly color-coded. Make module names clickable – clicking "Module 2" opens a detail view (see Module Checklist below) [44] [45]. - **Navigation Cards/Links:** Provide quick access to main sections: e.g. a card or button for "Document Editor" (perhaps opens the last opened doc or a new doc interface), "Template Library", "Regulatory Search", and "Export Submission". These can be stylized as tiles or a sidebar menu. - **User Menu:** Show the logged-in user's name/org, with a menu to logout or profile settings.

Component structure: We could have `Dashboard.jsx` to compose these. For instance, `RecentDocuments.jsx` and `ModuleProgress.jsx` as child components [46] [45]. On load, Dashboard calls APIs to get recent docs and progress stats.

2. Document Editor Page – The core of the app where users spend most time. This page embeds the Google Docs editor and provides side panels for AI and metadata [47] [48]. Components needed: - **Editor (Google Doc Iframe):** We use an `<iframe>` to embed the Google Doc. The src URL is obtained from backend (from GET `/documents/:id`). For example:

```
<iframe
  src={doc.embed_url}
  width="100%" height="100%"
  style={{ border: 0 }}
  title="Google Docs Editor">
</iframe>
```

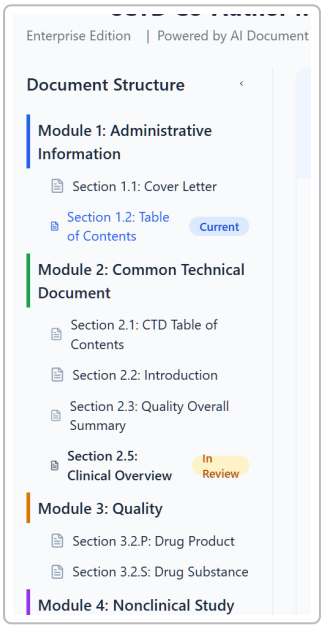
This will display the Google Docs UI inside our app. Because the user already went through Google OAuth, their Google session should be active. If not, the iframe will prompt them to log in to Google. We may need to ensure third-party cookies are allowed for Google or instruct the user accordingly.

```
Surrounding this iframe, we can have a top toolbar and sidebars.
```

- **Custom Toolbar:** At the top of the editor page, add a small toolbar with system-specific actions not provided by Google. For example, buttons like “Insert Template”, “AI Assist”, “Save Version”, “Submit for Review”, etc. [49]. This can be `EditorToolbar.jsx` [50]. Some of these buttons will call our backend or trigger UI modals:
  - *Insert Template:* Opens a dialog with template options (perhaps showing Templates by section). On selection, we could insert the template content. In Google Docs, inserting content might require using the Docs API to append text. Alternatively, if templates are small, the AI can also be asked to fetch and insert it. A simpler approach: open the template in another tab for copy-paste. But since we want integration, using the Docs API’s batchUpdate (with insertText requests) could place the template text at current cursor.
  - *AI Assist:* Toggles the AI Suggestions sidebar.
  - *Save:* (if we want manual save to vault) calls `/documents/:id/save`.
  - *Workflow actions:* If user’s role permits, show a “Submit for Review” or “Approve Final” button. When clicked, call `/documents/:id/state` and then perhaps redirect or update UI to reflect new status.
- **AI Assistant Sidebar:** A collapsible sidebar panel (`AISuggestions.jsx`) that shows AI outputs [48]. When open, it can have tabs or modes: one for "Suggest" (which might continuously show suggestions as you type or on demand), another for "Validate" (show list of issues after calling AI validate), and maybe "History" of prompts. For implementation, it could have a text area or just display suggestions with buttons to apply them. If a suggestion is shown, clicking it could insert the text into the Google Doc. However, programmatically inserting into Google Doc is non-trivial. One approach: copy the suggestion to clipboard and prompt the user to paste, or use the Docs API if we have the document ID and an authorized token (we do). The Docs API can insert text at a given

location if we know the range. Simpler: user places cursor and clicks “Insert Suggestion” – we call a backend function to do a Docs API insert at the end of the doc or prompt user to paste.
    - For the first iteration, it’s acceptable to just display the suggestion and let the user copy-paste it into the doc.
    - Validate tab: show a list of compliance findings from AI. Just display them for user to manually address.
    - Style the sidebar with Tailwind, e.g. a fixed width panel on right with overflow scroll for content, and a close button.
- **Metadata/Info Panel:** Either in a sidebar or on top, display document metadata like title, section, status, version, and maybe a link to open the DocuShare record or version history. This is mostly read-only info for context [51]. Could be part of a header or a collapsible panel.
- **Comments Panel:** (Optional) If implementing comments outside Google. But since Google Docs already has commenting, we might rely on Google’s system for inline comments. We could still have an overview of unresolved comments count in our UI.
- **Version History Modal:** (Optional in UI) A modal to show past versions (from DocumentVersions and DocuShare) [52]. The user could open it to download or view an older version. This modal triggers a backend call to get version list and displays them with dates and links.

Manage the state of these UI elements (sidebar open/closed, etc.) in React state. Use React Router such that navigating to `/editor/:docId` loads this Editor page. On mount, fetch document data from backend, then embed the iframe. Possibly subscribe to any events (though most editing happens within the iframe via Google, outside our React control).



A screenshot of the TrialSage Document Structure tree (left panel) shows Modules 1–5 and section statuses (e.g., “Current” for 1.2 Table of Contents, “In Review” for 2.5 Clinical Overview). This reflects how the UI presents the eCTD hierarchy to the user. The document tree can be part of the Editor page as a left sidebar or part of the dashboard module detail. In our design, we might include a collapsible left panel listing all sections in the current project’s CTD hierarchy (with icons or labels for status) so users can navigate between documents easily. Each entry can be clickable to open that document in the editor. This tree should be generated from a combination of template definitions and actual Documents present. For example, show all expected section

headings; if a document exists for that section, mark it (and show its status), otherwise allow user to create it by clicking a “+”. This gives a clear picture of completeness. The UI in the screenshot uses color bars for modules and icons for document status, which we can mimic with Tailwind styling (e.g., blue left border for Module 1, green for Module 2, etc., and badges like “Current” or “In Review” next to section names).

3. Template Library Page – A page or modal where users can browse available templates and read instructions for each. This can be a simple list grouped by module or region [53]. Use an accordion or list group UI with Tailwind for each module, listing templates (e.g., “2.5 Clinical Overview Template”, “2.7.1 Summary of Biopharm Studies Template”, etc.). Provide a button “Use Template” that would either create a new document from it (calls POST `/documents` with template_id) or insert into an open document. TemplateLibrary can be its own route (`/templates`) or a modal on the Editor page triggered by “Insert Template”. Implement it as a functional React component that fetches templates (`GET /templates`) on load and displays them. Only Admin sees controls to add/edit templates.

4. Regulatory Search Page/Panel – A tool for searching across prior submissions or a knowledge base [54] [55]. This can be either a full page (`/search`) or a slide-out panel on the editor (so users can quickly lookup something without leaving the doc). For implementation ease, consider a page with a search bar and results list. The results come from `GET /search?query=...` (Step 4). Display results with enough context: maybe the document title, a snippet of text around the found term, and if it’s from a prior project, indicate which project. If using vector search for semantic matching, we could also show “suggested references”. Each result could have an “Insert” button to grab that snippet into the current document. To do that, the frontend might copy text to clipboard or store it in a state, then the user can paste. (Advanced: directly use Docs API to insert, similar to AI suggestions, but clipboard is simpler.) Additionally, we can allow a question mode: user asks a question like “What is the recommended format for an EU Module 1?” – our backend could use GPT-4 with the search results (retrieval augmented generation) to answer. This is a stretch goal, but aligns with the idea of a "Knowledge Base" search [24]. For now, implement basic text search.

5. Export Wizard UI – A page or modal to guide the user through exporting a submission [56] [57]. This appears when user clicks “Export Submission” on the dashboard. Steps: - Step 1: Select Region (if multiple) and sequence number. If the project already has a region set, just display it. Sequence might default to 0000 or next available (you could fetch from SubmissionExports count). - Step 2: Select which modules to include. By default all, but user can uncheck optional sections or modules. Display a tree or checklist of modules/sections, with those without content disabled or marked. Ensure required docs are included. - Step 3: Confirmation – show summary (X documents will be included) and a “Generate” button. - After clicking Generate, call the backend and show a loading indicator (perhaps a spinner with “Generating package...”). - On success, present a download link or automatically download the file. Also list any validation warnings (if the backend returned some, like “Note: 2 documents were still Draft and not included”).

Use a multi-step form approach. This can be one component with internal state for step, or multiple subcomponents. Tailwind can style a progress indicator or just change content. The result (download link) could be just an anchor to the ZIP file path or a button that triggers another GET to download.

Use context or props to pass the current project and user info to all these components as needed. Also consider using React Router for navigation between pages (Dashboard, Editor, Templates, Search, etc.). Set up routes in App.jsx:

```
<Routes>
  <Route path="/" element={<Dashboard />} />
  <Route path="/editor/:docId" element={<EditorPage />} />
  <Route path="/templates" element={<TemplateLibrary />} />
  <Route path="/search" element={<SearchPage />} />
  <Route path="/export" element={<ExportWizard />} />
</Routes>
```

And ensure that after login (OAuth callback), the user is redirected to "/". We might have a simple login screen if not logged in (though with Google OAuth, you could also initiate login from backend).

Tailwind CSS styling: Use Tailwind classes to make it look clean. For example, use `bg-gray-100` for page background, `text-sm text-gray-700` for text, etc. The module progress bars can use Tailwind progress or a simple `<div style={{ width: percent% }} className="bg-green-500 h-2" />`. The left Document Structure tree can use list styling with indentations; use icons (maybe import a doc icon or use emoji) for document nodes. Badges like "Draft" or "Final" can be spans with Tailwind classes (`px-2 py-0.5 text-xs rounded bg-blue-100 text-blue-800` for example).

Focus on **responsiveness** minimally – the app will mostly be used on desktop, but ensure components don't overflow on narrower widths (Tailwind's grid or flex utilities can help). Add ARIA labels for accessibility on interactive elements.

At this point, the frontend should allow the user to: select a project, see status of all sections, create new documents (opening them in Google editor), get AI help, manage templates, search references, and export the package. This completes the interactive portion of the system.

Step 11: Multi-Region Template Handling and Region-Aware Document Tree

Supporting multiple regions (FDA, EMA, PMDA, etc.) means the system must adjust the available sections and templates for Module 1 and any regional differences [3]. We address this via configuration and dynamic UI:

- **Template Library by Region:** In the Templates table, each template is tagged with a region or "global". For Module 2–5 documents, templates can be common to all regions. For Module 1 (Administrative), each region has its own set of documents (e.g., FDA has Form 1571, EMA has an application form, etc.). At project creation, we assign a region. The frontend should filter templates to that region (or global) when showing options [58]. For instance, if project region is FDA, show templates for FDA Module 1 (like "1.1 Cover Letter (FDA)") and hide EMA-specific ones.

- **Document Structure Tree:** The left-hand document checklist should be generated based on region. We can maintain an object or JSON file that lists expected sections by region. For example:

```
const CTD_STRUCTURE = {
  "FDA": [
    { module: 1, title: "Module 1: Administrative Information", sections: [
        { code: "1.1", title: "Cover Letter" },
        { code: "1.2", title: "Table of Contents" },
        // ... etc, specific to FDA
      ]},
    // Modules 2-5 common
    { module: 2, title: "Module 2: Common Technical Document Summaries", sections: [ ... ] }
    ...
  ],
  "EMA": [ ...similar, with differences in Module 1... ]
};
```

This can be stored in a JSON file or database, but a static config in frontend is fine unless we want to allow admins to modify it.

When rendering the Document Structure, iterate through CTD_STRUCTURE for the current project's region. For each section, check if a document exists in the Documents list (which we fetched). If yes, display it with status; if not, display it greyed out or with an "Add" button. This guides the user on what documents need to be prepared for a complete submission [59].

Region differences: e.g. Module 1 for EMA might have "1.2 Application Form" whereas FDA has "1.2 TOC" and Form 1571 separately. Our config should handle that. The rest of modules are ICH common, so no differences there aside from perhaps some optional sections.

- **Region in Export:** The export logic (Step 9) should also consider region. If FDA, include any us-regional.xml specifics; if EMA, perhaps a different list of Module 1 docs. We already tailor content by only having what's in the project.

- **Form 1571 example:** FDA's Module 1 includes Form FDA 1571 (Investigational New Drug Application form). The design mentions possibly handling this via a special form or attaching a PDF [60]. Implementation: We could treat it as a template (a PDF template). Perhaps just note in UI that "Please upload Form 1571 PDF". An admin could have uploaded it to DocuShare outside the system, and we attach it. Due to time, we may not implement a PDF form filler, but we can allow uploading of an external file for such cases.

- **Reusability across regions:** If a company is authoring for multiple regions, content reuse is key (e.g., using the same Module 2 content for FDA and EMA submissions). Our design can support separate projects per region, but we could facilitate copying documents from one project to another or marking a document as "common". For now, keep it separate per project, but note this could be extended.

By handling region-specific templates and structure, we ensure the system can generate region-appropriate submissions. This draws inspiration from Celegence's Dosscriber approach of separating content from context for reuse across regions [61] [62]. We allow content to be largely the same in Modules 2–5 while

slotting it into different Module 1 wrappers for each region's requirements. The UI always guides the user on required documents per region, reducing the chance of missing something in, say, a JP PMDA submission that might not have been needed for FDA.

Step 12: Validation Dashboard and Completeness Checks

To ensure quality and compliance, implement a **Validation Dashboard** that checks the completeness of the current submission and highlights any issues. This is both a manual checklist and an AI-assisted check:

- **Completeness Check (Rule-based):** The system can automatically verify if all required sections for the project's region have been addressed. For example, if 10 documents are expected (per the CTD structure config) and only 8 are Final, list which ones are missing or still in Draft. This can be part of the Module Progress component or a separate page. On the Dashboard, the Module Progress bars already indicate completeness percentage. We can add a "View Checklist" that shows each required section with a status: Done (Final), In Progress (Draft/Review), or Missing. This module detail view (ModuleDetail.jsx) was considered in the design [63]. Implement it as either an expanded list under each module on click, or a dedicated page. It gives a quick way to see what work remains.

- **AI Compliance Check:** Extend the **/ai/validate** functionality to possibly run a project-wide check. For instance, after compiling, call GPT-4 with the index of documents or any summary to see if something obvious is missing. However, GPT-4 is more useful on individual docs. Instead, consider using it on Module 1 (do all forms exist?) or on a high level. This might be overkill; rule-based is sufficient to ensure no section is empty.

- **Validation Dashboard UI:** Could be a page summarizing:

- Missing documents: e.g., "No document for section 2.6.7" – user can click "Create" next to it.
- Documents still Draft/Review when expecting Final for submission: list those and perhaps action "Request finalize".
- Potential format issues: e.g., "The following documents have no PDF generated yet" or "Missing Table of Contents in Module 1" – though if following templates, TOC should be auto for CTD.
- If we had an internal checklist (like "All documents have correct headings for PDF bookmarks"), we could list that too. Some can be programmatically checked (like if using consistent styles – but since we rely on templates and Google, that might be fine).

The validation page essentially ensures the submission is **complete and consistent** before export. It's a safeguard so that when the user clicks export, they ideally have no missing pieces.

- **During Export:** Also incorporate validation – our export routine can fail or warn if something is missing. For instance, if a required doc is Draft, we could still include the last saved version but warn the user. We might output a list of warnings after generation that the UI shows.

By providing these checks, we reduce the chance of regulatory rejection due to an incomplete dossier. This addresses compliance at every step, echoing the design goal of ensuring ICH/FDA compliance throughout the process [64]. Writers can use the dashboard to track progress (e.g., Module 5 might be 45% complete, meaning many study reports not drafted yet) and plan accordingly.

Step 13: Searchable Submission History and Reference Integration

Implement features to leverage past submissions and institutional knowledge:

- **Submission History:** As we log exports in a SubmissionExports table, we can present a history of past submissions for the project (e.g., sequence 0000, 0001, etc., with dates). The Export Wizard might list previous exports as links [65]. More broadly, across projects, an Org Admin might have a view of all submissions made. This history ensures traceability of what was submitted when (useful for referencing previous sequences or resubmissions).

- **Full-Text Search:** Using the `/search` endpoint (Step 4), enable users to search across all documents they have access to (which could include current project and maybe other projects if within same org). This helps in reusing wording from prior approved documents. To implement effectively:

- When a document is finalized, use Google Docs API or DocuShare to get a text version of it and index that in the database. For example, we could store a plaintext or HTML copy of each Document in a column in DocumentVersions. Supabase supports full-text search on text columns; we can create a tsvector index for quick search.
- The search API will query that text. A simple SQL: `SELECT doc_id, title, ts_headline(text, query) AS snippet FROM DocumentVersions, to_tsquery($1) AS query WHERE text @@ query AND version_flag = 'Final';`. Alternatively, we can incorporate Supabase's built-in full-text search function in the JS client.
- Return top matches with maybe a snippet of context (we can highlight the term by using ts_headline as shown).

- On the frontend Search page, display these results. If a result is from a different project and the user has access, show project name. If they click it, maybe allow them to open it (either view the PDF or if it's their own previous submission, open read-only).

- **Prior Document Reference Injection:** This is a powerful feature where a user can insert a chunk from a previous document. The simplest approach: after searching, the user can copy the snippet from the results. We can ease this by providing an "Copy" button next to the snippet. In modern browsers, use the Clipboard API:

  ```
  navigator.clipboard.writeText(result.snippetText);
  ```

  Then the user can paste into the Google Doc. Alternatively, if we wanted to automate, we could again leverage Docs API to insert it at cursor, but that's complex for snippet placement. Copy-paste is acceptable.

- **AI + Search integration:** As an advanced capability, allow the user to ask a question and have the system answer using content from prior documents. For example, "Have we previously written a phase I trial summary for a similar compound?" The backend could then:

- Use the question to search the database for relevant docs.
- Take the top results (text content) and feed them along with the question into GPT-4 (this is known as Retrieval-Augmented Generation).
- Return the AI's answer, citing which document it drew from.

This could be surfaced in the UI as part of the search panel ("Chat with past submissions" feature). While not required, it aligns with making a smarter knowledge base. The design hints at this: *"the system uses GPT-4 with retrieval to answer based on the indexed data"* [24] .

Implementing basic search first is priority. Ensure the search obeys permissions – only returns documents from the user's org. If using Supabase's policy, that will naturally filter if we query a view limited by org.

Testing the search: Add some known text in two documents, finalize them, then search for a keyword present in one – verify that result appears with a snippet. Fine-tune snippet length for readability.

By enabling search and reuse, we tap into the cumulative knowledge of past submissions, which is invaluable. This addresses the need for a "Regulatory Search" and reference injection feature mentioned in the requirements, and is inspired by real industry tools (e.g., the search engine in Veeva or others) and by the vision of using AI to quickly retrieve prior relevant text [24] .

Step 14: Security and Multi-Tenancy Considerations

Security is paramount for a system managing sensitive regulatory documents. We address both **data security** and **access security**:

- **Authentication & Session Security:** All API calls require a valid JWT or Supabase auth session [66] . Use HTTPS (Replit provides SSL by default). Set secure cookies if cookies are used. On the frontend, guard routes so that if not logged in, user is redirected to login. Use short-lived tokens for sessions if possible and refresh them regularly (or rely on Supabase's JWT rotation). The Google and DocuShare tokens are stored server-side and never exposed to the client (except the initial OAuth redirect flow). This prevents XSS from stealing tokens.

- **Authorization & Data Isolation:** We implemented RBAC (Step 7) and RLS in the database (Step 2) to ensure users only access what they should. Multi-tenant segregation is enforced by scoping queries to `org_id`. Supabase's row-level security ensures even if a malicious query is attempted, cross-org data won't be returned [67] . Additionally, each organization can only see their projects. If using a single Google API key for all, that's fine, but note: we might be creating Google Docs under each user's account. If one user leaves an org, their docs remain in that user's Google Drive unless transferred. A more secure enterprise approach might have all docs under a service account or a dedicated Google Workspace domain. For now, our design assumes each user's Google account holds the docs they create (or we share a service Google account's docs with them). If using a service account, ensure that docs are segregated by project (maybe separate folders per org/project on Google Drive, though not strictly necessary as DocuShare is our system of record).

- **Encryption:** Store sensitive tokens (Google refresh token, DocuShare token) encrypted at rest. Since Supabase is Postgres, we can encrypt them before saving (e.g., using Node crypto with a secret). At minimum, mark those fields as varbinary and encrypt in code, or rely on Supabase's encryption

functions. Also encrypt any sensitive content we might cache (though we primarily store content in external systems and references).

- **Supabase Security:** If we use Supabase client from frontend for some direct access (we mostly don't in this plan, using our Node as middleman), ensure policies are set. In our approach, the backend uses the service role key (full access) but we trust our own auth checks. This means our Express server is a gatekeeper. That's fine as long as our code is correct. Alternatively, we could have used Supabase's JS library on frontend to directly do CRUD on docs with RLS, but then integrating Google and DocuShare from frontend is not possible (due to CORS and security), so the server approach is justified.

- **OpenAI API usage:** Rate limit the /ai/ endpoints to prevent abuse or accidental spamming (since each call costs $$$). We can implement a simple cooldown or limit calls per minute per user. Also, do not log or expose any PHI or sensitive data to OpenAI beyond what's necessary (the documents likely contain confidential info, but it's expected by using GPT-4 we are sending text to OpenAI's servers – make sure the org is aware and consents to that, as per OpenAI policy and any privacy requirements). Perhaps mention this in a security note or allow turning off AI for ultra-sensitive projects.

- **Audit Trail:** Keep logs of user actions – e.g., state changes, exports, logins – in an audit table. This is often needed for compliance. Even a simple text log file on Replit or entries in a Supabase `AuditLog` table (with timestamp, user, action, details) can help trace any changes.

- **Document Storage:** We use external systems for documents, but note that Replit's file system is not intended for persistent private storage beyond the container. We use `/mnt/data` for temp files (like PDFs, zips) which persists across restarts but is not intended as long-term storage for user data off Replit. After generating a zip, either clean it up after some time or store it in Supabase Storage which is more persistent and secure (with access controls).

- **Preventing Data Leakage:** When embedding the Google Doc, ensure that the user only sees docs they have rights to. For instance, if using a service account and a generic embed link, *don't* make the doc publicly accessible, as that could leak content. Instead, use Google Drive permissions: share the doc explicitly with the user's Google account on creation (Drive API `permissions.create` to give the user or a group read/write access). This way, the embed will only work for that user. If multiple authors, share with all collaborators. Revocation: when someone should no longer access, remove their permission.

- **Testing multi-tenant isolation:** Create two dummy orgs with users, create docs in each, and verify that one org's user cannot access the other's docs via API (should get 403 or no results). Also test UI shows only relevant data.

- **Using Replit safely:** Replit's always-on instance will keep the server running. Monitor memory/CPU usage especially when generating PDFs or zips. If needed, offload heavy work (like PDF generation) to asynchronous tasks or on-demand to avoid blocking the server.

By following these security practices, we ensure the system protects confidential submission data and maintains compliance. The combination of OAuth2, RBAC, and vault integration provides a robust security

posture: OAuth2 ensures secure access to external services [12], RBAC and RLS protect internal data, and DocuShare serves as a secure repository with its own access controls and audit logs [4]. This way, TrialSage eCTD Co-Author can be safely used by multiple teams or even multiple organizations without risk of data crossover or unauthorized access.

Step 15: Deployment and Testing on Replit

With all components built, we deploy and test the system on Replit:

- **Running the App:** Configure the Replit run command to start the Express server. If serving the React build from Express, ensure you have built the frontend (`npm run build` in the `/frontend` directory) and the build output (static files) is accessible. For example, you can have Express serve the `frontend/build` directory:

  ```
  app.use(express.static(path.join(__dirname, '../frontend/build')));
  app.get('*', (req, res) => {
    res.sendFile(path.join(__dirname, '../frontend/build/index.html'));
  });
  ```

  This will handle routing in the client. Alternatively, during development you might run the React dev server on a different port and proxy API calls. In Replit's single container, it's simpler to serve the built files as above for production mode.

- **Environment Variables on Replit:** Double-check all required secrets are added via the Secrets tab. Replit will set them in process.env for our Node server. For the React app, if it needs any config (like Supabase URL or so), you might use a .env file in frontend and the Replit Secrets can inject at build time. But since we handle most through backend, the frontend might only need perhaps the Supabase anon key if using Supabase client directly for auth. If we avoid direct Supabase calls from frontend, then just ensure backend knows all secrets.

- **Testing OAuth on Replit:** Replit's hosted URL might not be on a public allowlist by Google by default. Make sure the OAuth consent screen is configured (Google Cloud Console) and that the redirect URI exactly matches including https and no trailing slash. Try the Google login flow: it should redirect and then land back on our callback and then our app. On success, ensure the session is set (maybe check cookies or localStorage for token). Test that the iframe loads the Google Doc (this might prompt to login the first time inside the iframe – if so, log in and it should then display the doc content). This part may require some finesse: if issues arise with embedding (e.g., Google might block if the app isn't verified or if third-party cookies blocked), an alternative is to open the doc in a new tab. But assume we got the embed working as per design.

- **Testing DocuShare integration:** Use a sandbox or test account on DocuShare if possible. Test the login, create doc, upload version, retrieve version. If DocuShare isn't readily testable, mock those API calls during development to not block overall testing. Ensure the logic doesn't break if DocuShare is offline (maybe queue operations to retry).

- **AI Testing:** Use a small text and call /ai/suggest to see if we get a response from OpenAI. Because GPT-4 can be slow, consider using GPT-3.5 for quick tests (just change model to `gpt-3.5-turbo` if using ChatCompletion). Once confirmed, switch to GPT-4 for better quality. Check the suggestions and validate output for reasonableness and that our prompt formatting is okay.

- **End-to-End Scenario:** Walk through a full scenario:

- Login with Google (and DocuShare if needed).
- Create a new Project (if we allow via UI or insert one in DB).
- On Dashboard, click "New Document" for, say, Module 2.2 Introduction. Pick a template. Document opens in Editor.
- Write some text, use "AI Suggest" – see suggestion and manually insert it.
- Save or just directly Submit for Review.
- As a Reviewer (perhaps simulate by just having same user or change role in DB), open document, use AI Validate – see any issues.
- Mark as Final. Ensure PDF is saved and DocuShare updated.
- Go to Export, generate package. Download ZIP and inspect its contents: the PDF should be present and index.xml listing it.
- Try search: search for a keyword in the text just written, see that it appears (if we indexed current docs; if not, finalize then it would be indexed).

- Try multi-tenant: create another org or project, ensure documents from first don't show.

- **Monitoring:** Since this will run on Replit, monitor logs (Replit console) for any errors. Replit's always-on service should keep it up, but if memory leaks or crashes occur, we may need to optimize.

- **Iteration:** Fix any issues found in testing (e.g., adjust CORS if frontend calls aren't reaching backend – likely need `app.use(cors())` with appropriate origin). Also ensure the Supabase connection is stable; on free tier, connections may drop – consider reconnect logic or using Supabase's HTTP API instead of direct PG connection.

Once everything is working end-to-end on Replit, document how to use the system for end users. Include steps like "Connect your Google account, then choose a section to start writing…", but that's beyond build scope.

This completes the development playbook. Following these steps, we have set up a comprehensive eCTD co-authoring platform, inspired by industry solutions (like Certara's AI CoAuthor and Celegence's Dosscriber templates) but implemented on an agile stack that even a small team can deploy [68] [69]. The final system running on Replit demonstrates modern web tech (React, Node, cloud APIs) orchestrating to make regulatory writing faster, smarter, and more collaborative than ever.

Intelligent eCTD Co-Author System Design Overview (1).pdf
file://file-DNkLRfEstKPFwU5YocACcD

TrialSage eCTD Co-Author Module_ Enterprise-Grade Upgrade Design.pdf
file://file-6aC1Fy8PFeQN1RZJZ1uWzG