

Issue:

When I click “Create New Project” or “All Projects”, both actions are leading to a 404 error page. Each of these buttons is supposed to navigate to their specific, valid routes, but instead they are both routing incorrectly.

Expected Functionality

1. Create New Project

When the user clicks “Create New Project”, the app must open a Templates Page.

This Templates Page should display all available project templates.

When the user selects a template:

A new project should be created automatically based on that template.

Then it should redirect to the project workspace/editor for that newly created project.

2. All Projects

When the user clicks “All Projects”, they must be taken to the Projects List Page.

This page should display:

All existing projects

Options to open, edit, rename, or delete a project

No 404 pages should appear.

What Needs to Be Fixed

Ensure both buttons route to their correct paths.

Fix or create the required endpoints/pages:

/projects/new → Templates Page

/projects/new/:templateId → Auto-create new project → redirect to workspace

/projects → All Projects Page

Remove or fix the incorrect routes causing the 404 errors.

Make sure template selection triggers the correct backend call for project creation.

What Claude Should Do

Reimplement the routing logic.

Ensure proper frontend navigation.

Ensure backend endpoints exist and respond correctly.

Implement robust error-handling so that 404s do not occur unless the route truly does not exist.

Confirm the user flow from:

Dashboard → Create New Project → Select Template → Workspace

Dashboard → All Projects → Projects List Page

You are the technical architect and senior full-stack engineer responsible for implementing the Notes section in a Next.js/React application.

Your goal is to implement a Microsoft Word-like rich text editor structured into Jupyter-style blocks with a global ribbon and per-block rephrase actions.

Follow all specifications strictly. Maintain consistency in UI/UX and adhere to the API structures and behaviors described below.

 FULL TECHNICAL SPECIFICATION (Claude should follow this)

1. Overall Behavior

The Notes tab is a section containing multiple editable blocks.

Each block contains:

A header

A rich-text area

A Rephrase button

Users can:

Add new blocks

Delete blocks

Rearrange blocks

Type rich text

Paste images / screenshots

Use a global formatting ribbon (bold, italic, bullets, etc.)

2. GLOBAL RIBBON (One toolbar for formatting)

Must include:

Bold

Italic

Underline

Heading/Font size

Bulleted list

Numbered list

Checkbox list

Toggle bullets

Insert image (upload)

Undo/Redo

Highlight and text color (optional)

Behavioral Rules:

Ribbon affects selected text in the currently focused block.

Works exactly like Microsoft Word formatting.

No per-block mini toolbar.

Use TipTap, Slate.js, or Quill with custom extensions for image paste.

3. BLOCK BEHAVIOR (Critical)

Each block behaves like a Jupyter Notebook cell:

Actions:

Add new block (button or hit Enter at end)

Delete block (button or empty block Backspace)

Move blocks up/down (optional)

Each block has:

Header (editable, default = "New block header name")

Rich text area

Rephrase button (LLM action)

Image Support:

Paste screenshots directly (Ctrl+V)

Drag + drop images

Upload via ribbon button

4. REPHRASE FUNCTIONALITY

Action:

Triggered by clicking "Rephrase" on a block.

Behavior:

Extract only the text content inside that block.

Send to LLM with instruction:

“Rephrase this text without changing meaning or context. Improve grammar, clarity, and human tone.”

Replace block content with rephrased version.

Images in that block must not be touched.

5. COMPONENT ARCHITECTURE (React / Next.js)

/components

/Notes

NotesContainer.jsx

Ribbon.jsx

Block.jsx

BlockHeader.jsx

BlockEditor.jsx

RephraseButton.jsx

NotesContainer

Holds an array of blocks

Handles add/delete/reorder logic

Passes appropriate handlers to children

Ribbon

Global toolbar

Controls formatting using editor commands via TipTap/Slate

Block

Parent wrapper for header, editor, rephrase

Has unique blockId

Receives callbacks (onDelete, onRephrase, onMoveUp, onMoveDown)

BlockHeader

Editable header (contenteditable div or input)

BlockEditor

Rich-text editor instance

Handles paste, images, selection, formatting

RephraseButton

Calls backend or LLM API

Updates block text

6. DATA MODEL / JSON STRUCTURE

Frontend Block Object:

```
{  
  "id": "uuid",  
  "header": "New block header name",  
  "content": "<p>Rich text in HTML or a JSON editor format</p>",  
  "order": 1  
}
```

Notes Document:

```
{  
  "noteId": "uuid",  
  "blocks": [ ... ],  
  "lastUpdated": "timestamp"  
}
```

7. API DESIGN

POST /api/notes/:noteId/rephrase

Body:

```
{  
  "blockId": "uuid",  
  "text": "original block text"  
}
```

Response:

```
{  
  "rephrasedText": "improved text"
```

}

PUT /api/notes/:noteId/blocks

Update the whole block list.

8. WORKFLOW LOGIC

Adding a block:

At end of block: user presses Enter → new block created

Or user clicks "Add Block" button

Deleting a block:

User clicks delete icon

OR block is empty and user presses Backspace → remove block

Pasting image:

Detect paste event containing file/blob

Insert image node into TipTap/Slate tree

Rephrasing:

Disable block

Send request

Replace content

Re-enable block

9. KEY EDGE CASES

Empty block + Backspace → delete block

Images inside block must NOT be removed or edited by rephrase

User must not lose focus after rephrasing

Formatting must not reset

Blocks must keep order consistent

10. IMPLEMENTATION CHECKLIST (Claude should follow)

Editor & Formatting

✓ Word-like formatting

✓ Only one global ribbon

✓ Per-block selection awareness

Blocks

✓ Add block

✓ Delete block

✓ Editable header

✓ Rich text body

✓ Paste images

✓ Drag reorder (optional)

Rephrase

✓ LLM API

✓ Prevent meaning change

✓ Preserve images

✓ Update text only

Persistence

✓ Store blocks array in DB

✓ Save on every update or with debounce

11. INSTRUCTIONS TO CLAUDE (Put this when prompting Claude for engineering tasks)

Always generate code and logic following:

Component architecture provided above

Data model provided above

API endpoints provided above

UX rules (Word-like formatting, Jupyter-like blocks)

Rephrase logic exactly as written

Only global ribbon

Images must be supported

FINAL SPECIFICATION: AI Research Assistant (With Greeting + Chat History)

1. Overview

The AI Research Assistant must support:

A default greeting message

Persistent chat history

A clean “New Session” workflow

The greeting must appear ONLY when the user starts a new session manually

✅ 2. Default Greeting Message

Default Message Text

"Hello ! I'm your AI research assistant . How can I assist you today ?"

When it Appears

ONLY when the user clicks the “New Session” button.

It does not appear:

On page refresh

On returning to an existing session

While loading old conversations

How to Insert It

Insert it as the first assistant message in that new session.

Must Be Stored

The greeting is stored in the conversation history database as the first message.

It is not a placeholder or UI-only message.

✅ 3. Chat History System

Requirements

Every session has:

```
{  
  "sessionId": "uuid",  
  "messages": [],  
  "createdAt": "...",  
  "updatedAt": "..."  
}
```

When user selects a previous session:

Load its messages

Do NOT add greeting

When user creates a new session:

Create a new sessionId

Insert the default greeting as the first message

Wait for user's first message afterward

Message Format

Each message follows this structure:

```
{  
  "id": "uuid",  
  "role": "user" | "assistant",  
  "content": "string",  
  "timestamp": "ISO string"  
}
```

✓ 4. UI/UX Behavior

New Session Flow

User clicks “New Session”

Clear chat window

Insert the default greeting message

Scroll to bottom

Focus input box

Existing Session Flow

Load all previous messages

Do NOT show greeting

Refresh Page

Must reload the last used session

Do NOT add greeting

✓ 5. Backend Logic

Creating a New Session

POST /api/research-assistant/session

Create session document

Insert greeting message

Return sessionId + messages

Loading a Session

GET /api/research-assistant/session/:sessionId

Returns all messages

No new greeting added

Adding a User Message

POST /api/research-assistant/session/:sessionId/message

Adding Assistant Reply

Append to existing session

Update updatedAt

 6. Core Rules (Claude MUST follow)

Greeting appears ONLY on “New Session”

Greeting is a real message, stored in DB

No greeting on:

Page refresh

Returning to old session

Switching sessions

Chat history must remain untouched unless "New Session" is clicked

 7. Additional Notes for Implementation

If last active session is empty (no messages), auto-add greeting (optional but recommended).

Ensure timestamp consistency.

Avoid duplicate greeting creation in case of double-clicking “New Session.”

Additional information;

1) Quick UI wireframe (text)

Sidebar (left) Chat Window (right)

Search sessions / + New Session button (top) Chat header (session title) & actions (Delete session, Rename)

List of sessions (click to load) Message area (scrollable) — messages in order (assistant/user). On new session the first message is the greeting saved in DB.

Composer (text input) + Send button

New Session flow:

User clicks + New Session

API POST creates session + inserts greeting message

UI loads new session and shows greeting (assistant message) — focus composer

2) Folder structure (Next.js app dir or pages dir)

/app or /src

/components

/ResearchAssistant

ChatSidebar.jsx

ChatWindow.jsx

Message.jsx

NewSessionButton.jsx

SessionListItem.jsx

/context

SessionContext.jsx

/hooks

useSessions.js

useSessionMessages.js

/lib

db.js // Prisma client

apiHelpers.js

/pages (or app/routes)

/api

/research-assistant

sessions.js // POST list/create

[sessionId].js // GET session, DELETE session, PUT rename

[sessionId]/message.js // POST add message

/prisma

schema.prisma

/tests

researchAssistant.test.js

3) Database schema (Prisma / Postgres)

Prisma schema (schema.prisma) — minimal, durable:

```
generator client {  
  provider = "prisma-client-js"  
}  
  
datasource db {  
  provider = "postgresql"  
  url = env("DATABASE_URL")  
}  
  
model Session {  
  id String @id @default(cuid())  
  title String?  
  createdAt DateTime @default(now())  
  updatedAt DateTime @updatedAt  
  messages Message[]  
}  
  
model Message {  
  id String @id @default(cuid())  
  sessionId String  
  role String // "assistant" | "user"  
  content String @db.Text  
  createdAt DateTime @default(now())  
  session Session @relation(fields: [sessionId], references: [id], onDelete: Cascade)  
}
```

Notes:

Greeting is stored as first Message with role = "assistant".

Use @db.Text for long messages.

Mongo alternative: a sessions collection with { _id, title, createdAt, updatedAt, messages: [{ _id, role, content, createdAt }] }.

4) API endpoints (Next.js API route examples)

Routes overview

POST /api/research-assistant/sessions — create new session (creates greeting message)

GET /api/research-assistant/sessions — list sessions (lightweight)

GET /api/research-assistant/sessions/:sessionId — get full session with messages

POST /api/research-assistant/sessions/:sessionId/message — add user message; returns assistant reply if server generates it

PUT /api/research-assistant/sessions/:sessionId — update session (rename)

DELETE /api/research-assistant/sessions/:sessionId — delete session

Example: create new session (simplified Next.js API)

```
// pages/api/research-assistant/sessions.js (POST)
```

```
import prisma from '../../../lib/db';
```

```
const GREETING = "Hello ! I'm your AI research assistant . How can I assist you today ?";
```

```
export default async function handler(req, res) {
```

```
  if (req.method === 'POST') {
```

```
    // Prevent double creation on double click (simple idempotency token optional)
```

```
    const { title } = req.body || {};
```

```
    const session = await prisma.session.create({
```

```
      data: {
```

```
        title: title || "New session",
```

```
        messages: {
```

```
          create: [{ role: 'assistant', content: GREETING }]
```

```
        }
```

```
      },
```

```

include: { messages: true }
});
return res.status(201).json(session);
}
if (req.method === 'GET') {
const sessions = await prisma.session.findMany({
orderBy: { updatedAt: 'desc' },
select: { id: true, title: true, updatedAt: true, messages: { take: 1, orderBy: { createdAt: 'asc' } } }
});
return res.status(200).json(sessions);
}
res.setHeader('Allow', ['POST', 'GET']);
res.status(405).end(` Method ${req.method} Not Allowed `);
}

```

Example: get session messages

```

// pages/api/research-assistant/[sessionId].js (GET)
import prisma from '../../lib/db';
export default async function handler(req, res) {
const { sessionId } = req.query;
if (req.method === 'GET') {
const session = await prisma.session.findUnique({
where: { id: sessionId },
include: { messages: { orderBy: { createdAt: 'asc' } } }
});
if (!session) return res.status(404).json({ error: 'Session not found' });
return res.status(200).json(session);
}
// handle PUT (rename) and DELETE as needed
}

```

Example: add message

```
// pages/api/research-assistant/[sessionId]/message.js (POST)

import prisma from '../../../../lib/db';

export default async function handler(req, res) {

  const { sessionId } = req.query;

  if (req.method !== 'POST') return res.status(405).end();

  const { role, content } = req.body;

  if (!role || !content) return res.status(400).json({ error: 'role and content required' });

  const message = await prisma.message.create({

    data: { sessionId, role, content }

  });

  // Update session updatedAt

  await prisma.session.update({ where: { id: sessionId }, data: { } });

  // Optionally: if you want the assistant reply generated server-side, call LLM here,

  // store assistant message and return it. For now return the user message.

  return res.status(201).json(message);

}
```

Important idempotency: for POST /sessions, add an optional client-provided clientId to avoid duplicates.

5) React client architecture & key snippets

State management approach

Use React Context (SessionContext) to store current sessionId + session data.

Use SWR or React Query to fetch sessions and messages with automatic revalidation.

Optimistic UI when adding message.

SessionContext (simplified)

```
// src/context/SessionContext.jsx

import React, { createContext, useState, useContext } from 'react';

const SessionContext = createContext();

export function SessionProvider({ children }) {
```

```

const [sessionId, setSessionId] = useState(null);
const [sessionTitle, setSessionTitle] = useState(null);
return (
  <SessionContext.Provider value={{ sessionId, setSessionId, sessionTitle, setSessionTitle }}>
    {children}
  </SessionContext.Provider>
);
}

export const useSession = () => useContext(SessionContext);

NewSessionButton (creates session and loads)

import { useRouter } from 'next/router';
import useSWR from 'swr';
import fetcher from '../lib/fetcher';
import { useSession } from '../context/SessionContext';
export default function NewSessionButton() {
  const { setSessionId } = useSession();
  const router = useRouter();
  const createSession = async () => {
    const res = await fetch('/api/research-assistant/sessions', { method: 'POST' });
    const session = await res.json();
    setSessionId(session.id);
    router.push(`/research-assistant/${session.id}`);
  };
  return <button onClick={createSession}>+ New Session</button>;
}

ChatWindow (load messages with SWR)

import useSWR from 'swr';
import { useSession } from '../context/SessionContext';
import Message from './Message';

```

```

import Composer from './Composer';

export default function ChatWindow() {
  const { sessionId } = useSession();

  const { data: session, error, mutate } = useSWR(
    sessionId ? `/api/research-assistant/sessions/${sessionId}` : null,
    fetcher
  );

  if (!sessionId) return <div>Select or create a session</div>;

  if (!session) return <div>Loading...</div>;

  return (
    <div className="chat-window">
      <div className="messages">
        {session.messages.map(msg => <Message key={msg.id} message={msg} />)}
      </div>
      <Composer sessionId={sessionId} mutate={mutate} />
    </div>
  );
}

Composer (send message)

export default function Composer({ sessionId, mutate }) {
  const [text, setText] = useState("");

  const send = async () => {
    if (!text.trim()) return;

    // optimistic update

    const tempMsg = { id: 'temp-' + Date.now(), role: 'user', content: text, createdAt: new Date().toISOString() };

    mutate(prev => ({ ...prev, messages: [...prev.messages, tempMsg] }), false);

    await fetch(`/api/research-assistant/sessions/${sessionId}/message`, {
      method: 'POST',

```

```

headers: { 'Content-Type': 'application/json' },
body: JSON.stringify({ role: 'user', content: text })
});
setText("");
mutate(); // revalidate to get saved message(s)
};
return (
<div>
<textarea value={text} onChange={e => setText(e.target.value)} />
<button onClick={send}>Send</button>
</div>
);
}

```

6) UX & Session logic details (must follow)

New Session:

Button disabled while request in-flight (prevent duplicates)

Server creates session + greeting message

Client navigates to session and shows greeting (first assistant message)

Focus composer

Page refresh:

Restore the last active session (persist last session id in localStorage)

Do not re-add greeting

Loading old session:

Just fetch messages, do not add greeting

Greeting storage:

Greeting is an actual message stored as role: assistant as the first message

Idempotency:

Optional requestId for POST /sessions; server should ignore duplicates for same token within a short TTL

7) Edge cases & testing checklist

Edge cases to handle (include unit + integration tests):

Double-click New Session → duplicate sessions: test prevention (disabled button or server idempotency).

Create session, close tab, reopen by URL → greeting appears exactly once.

Loading session with zero messages — behavior: should still show session (but in our logic every new session has greeting).

Race conditions when multiple tabs create sessions simultaneously.

Timezone/timestamp consistency (store ISO timestamps UTC).

DB failure on session creation — rollback behavior: if message insert fails, delete session.

Large message sizes — ensure DB column handles Text.

Tests:

API tests for POST /sessions adds greeting

GET session returns messages ordered ascending

Composer optimistic update -> server persisted

8) Security & infra notes

Validate user authentication — sessions should be scoped to user (add userId to Session model).

Rate limit session creation to avoid spam.

Sanitize message content before saving/display.

If generating assistant responses server-side (LLM), secure API keys and implement retry/backoff & logging.

9) Deliverables checklist for Claude/devs to implement

Prisma schema implemented and migrated

API routes created and tested

React components (Sidebar, ChatWindow, Composer) implemented

SessionContext & SWR hooks implemented

New Session button creates session and inserts greeting

Greeting stored in DB, visible as first message

LocalStorage store for last active session on refresh

Tests for create session / get session / add message

UX polish (focus, scroll to bottom, disable double clicks)

Documentation snippet for greeting logic (where it’s injected & stored)

✔ Literature Tab — Complete Functional Specification

1. Remove existing manual entry fields

Delete the previous manual fields (author, year, etc.).

User will now add a paper using the new unified flow below.

✔ 2. New “Add Paper” Flow

When the user clicks Add Paper, show a page/modal that contains only:

Required Inputs

Field	Type	Required	Notes
Title of the paper	Text	Yes	User manually enters it
PDF upload	File	Yes	Must upload PDF to proceed

Optional Inputs

Field	Type	Required	Notes
DOI	Text	No	Optional
Notes	Textarea	No	User may add personal notes about the paper

After user submits:

The PDF is stored.

A new literature entry is created in the database.

It is listed in the Literature Tab list on the left.

✔ 3. When clicking an added paper

When the user selects a paper from the list:

A right-side panel opens (same space where the user originally entered title/DOI/etc.).

This panel now becomes the AI Literature Assistant window.

This panel contains:

A) Header

Title of the selected paper

Optional: file name, DOI, delete/rename options

B) AI Literature Assistant

The AI chat window opens automatically with a default message:

Default Message (Improved English)

“Hello! Would you like me to give you a quick summary of this paper?”

✔ 4. Capabilities of the AI Literature Assistant

A) Paper-specific intelligence

The model receives:

The uploaded PDF content

Title

Notes (optional)

DOI (optional)

The AI must be able to:

Summarize the paper

Explain the methodology

Explain each figure / table (if requested)

Extract key findings

Explain formulas, equations, or models used

Relate the paper to the user's research topic

Give suggestions on how user can extend or use the methods

Provide clarity on experimental setup

Explain domain-specific terms

✔ 5. Internet-Assisted Literature Search (Important Feature)

Inside this AI assistant:

The LLM must have the ability to search the internet for:

Similar papers

Papers with the same research topic

Papers that cite the uploaded paper

Papers referenced by the uploaded PDF

Relevant reviews / survey papers

The assistant can also:

Provide abstracts of suggested papers

Give comparison summaries

Suggest next reading steps

Example user commands:

“Find similar papers on zinc–air batteries with PMMA electrolytes.”

“Give me abstracts of 3 papers that use the same characterization technique.”

“Find papers that improve this method.”

✅ 6. Architecture Summary for Developers

Database:

New fields per literature document:

id

title

doi (optional)

notes (optional)

pdfUrl

createdAt

updatedAt

Frontend Behavior:

Left side → list of uploaded papers

Right side → dynamic panel showing AI assistant

AI Assistant Context:

Send to LLM:

Extracted text from PDF

Title

DOI

User notes

Paper metadata

User's query

Internet Search Module:

Implement a search API wrapper (e.g., Bing API, Serper, Tavily, or custom scraper) that can provide:

List of similar articles

Abstracts

Citation suggestions

✓ 7. Additional Enhancements (optional but powerful)

Let me know if you want these added:

PDF preview inside the panel

AI-generated structured notes (Summary, Methods, Results, Limitations)

Auto-extraction of references from PDF

Citation graph view

Compare two papers side-by-side

Tagging & categorization for literature library

Auto-detect title & DOI from PDF metadata

1) High-level UX summary

Leftmost collapsible navigation strip (auto hide after 5s idle; small icon remains).

Main content area: Left Sidebar (Literature list + nested folders) and Right Panel (AI Literature Assistant & paper metadata).

Data & Analysis removed.

2) Left Sidebar — Papers + Nested Folders (Behavior & UI)

The sidebar lists folders and uncategorized papers.

Folders are collapsible toggle groups; each folder may contain:

Papers (leaf nodes)

Subfolders (nested; unlimited levels, but UX should limit deep nesting visually)

Folder row shows: folder name, expand/collapse chevron, count of items.

Paper row shows: title, small PDF icon, (optional) DOI shown on hover or expansion.

Clicking a paper opens the right panel with the AI assistant (and paper metadata).

Right-click or kebab menu on paper: Move → choose folder, Rename, Delete.

Drag & drop:

Support dragging papers into folders and subfolders.

Support dragging a folder into another folder (re-parent).

Visual placeholder for drop target and forbidden drop states (e.g., don't allow a folder to be dropped into itself or its descendant).

Breadcrumbs: when viewing a paper, show its folder path near the header (e.g., Battery Papers > Li-ion > Cathode).

3) Data model (DB)

Use nested set or parent reference with path / ancestry field for efficient queries. Example (Postgres + Prisma):

Folder

```
{
  "id": "uuid",
  "name": "string",
  "parentId": "uuid|null", // null for top-level folders
  "path": "/parentId/childId/...", // string of folder ids for quick ancestry checks (optional)
  "createdAt": "ISO",
  "updatedAt": "ISO",
  "ownerId": "uuid" // scope to user
}
```

Paper

```
{
  "id": "uuid",
  "title": "string",
  "doi": "string|null",
  "notes": "text|null",
  "pdfUrl": "string",
  "folderId": "uuid|null", // null = Uncategorized
  "createdAt": "ISO",
  "updatedAt": "ISO",
  "ownerId": "uuid"
```

}

Notes about path approach

Maintain path as "/rootId/parentId/thisId/" to quickly query descendants and prevent cycles.

On move, update parentId and recalculate path for moved folder and its descendants.

4) API endpoints (suggested)

GET /api/literature/folders — returns full folder tree (nested) and counts.

POST /api/literature/folders — create folder { name, parentId? } → returns created folder.

PUT /api/literature/folders/:id — rename or change parent { name?, parentId? }.

DELETE /api/literature/folders/:id — delete folder (option: cascade move children to parent or to Uncategorized).

POST /api/literature/papers — upload new paper { title, doi?, notes?, file } → returns paper record.

GET /api/literature/papers/:id — get paper metadata + PDF URL.

PUT /api/literature/papers/:id — update metadata / move folder { title?, doi?, notes?, folderId? }.

DELETE /api/literature/papers/:id

POST /api/literature/papers/:id/move — helper to move paper to folder { folderId? } (optional)

Add batch endpoints for drag-drop operations (move multiple items).

All endpoints must validate ownerId scope.

5) Frontend components (React)

/components/Literature

Sidebar.jsx // renders FolderTree + PaperList

FolderNode.jsx // recursive component for folder + children

PaperRow.jsx

Breadcrumbs.jsx

PaperRightPanel.jsx // AI assistant + PDF preview + metadata

SlideNavStrip.jsx // leftmost collapsible strip

FolderNode.jsx must be recursive: renders folder header then maps children.folders and children.papers. Provide depth prop to limit UI indentation and optionally collapse very deep branches.

6) Drag & Drop details

Use a robust library (e.g., react-dnd or @dnd-kit/core).

Rules:

Prevent dropping folder into one of its descendants.

Show visual cue when drop is allowed or disallowed.

When dropping, call API to update parentId or folderId.

For moving a large subtree, update path for all descendants on backend; frontend can optimistic-update.

7) Move / Delete behaviors (UX rules)

Deleting a folder: show modal with options:

Move all children (folders + papers) up to parent folder.

Move all children to "Uncategorized".

Delete everything (cascade).

Default behavior: move children up to parent (safe option).

Deleting a paper: confirm modal with "Delete" or "Cancel".

8) Slide-In/Slide-Out Left Navigation Strip (implementation)

Component: SlideNavStrip.jsx

Behavior:

On initial page load, show strip for 5 seconds, then slide out if no interaction.

Track user interaction events on the strip: mouseenter, click, focus — reset the 5s timer.

When slid out, show a floating icon at left edge. Clicking icon slides strip in and resets timer.

Use CSS transform `translateX(-100%)` for slide-out and `translateX(0)` for slide-in; use transition: `transform 350ms ease`.

Accessibility: keyboard focus should also toggle (e.g., pressing `Ctrl+B`? optional) and provide ARIA labels for collapsed state.

Persist strip preference in `localStorage` if user manually pins/unpins strip (optional: add a pin icon to keep it open).

9) Frontend performance & UX considerations

Lazy-load deep folder contents (expand only fetch children when opened) if user has thousands of items.

Virtualize long lists (e.g., `react-virtualized`) for performance.

For drag & drop across large trees, show a short loading skeleton while backend updates.

Provide undo toast for destructive actions (e.g., "Paper moved to 'Electrolyte' — Undo").

10) Edge cases & safeguards

Prevent circular parenting: backend must validate parentId is not a descendant of the folder being moved.

Limit nesting depth in UI (e.g., visually collapse beyond depth 6 and offer "View more" to avoid UI break).

Concurrency: handle simultaneous moves/renames with optimistic locks or last-write-wins and notify users if conflicts happen.

File storage: validate PDF upload (MIME type application/pdf), size limits, and virus scanning if necessary.

Permissions: enforce owner-only edits; consider workspace/team sharing later.

11) Tests to add

Unit tests for folder create / rename / delete / move operations (including path updates).

Integration tests for drag & drop behavior.

E2E tests covering: create folder, create subfolder, upload paper, move paper into nested folder, delete folder with cascade options.

UI tests for slide-in/out strip animation and persistence.

12) Developer checklist for Claude

Implement folder DB model with parentId + path

Implement APIs for CRUD and move operations with validations

Implement recursive FolderNode component with drag & drop

Implement PaperRow & right-panel load on click

Implement slide-in/slide-out navigation strip with timer & pin option

Remove Data & Analysis components, routes and nav items

Add unit, integration, E2E tests

Add safeguards (cycle prevention, nesting limit, file validation)

1 — High-level behavior (summary)

Users can select entire columns or a row-range (e.g., rows 3–10) to plot.

Row-range selection filters the dataset to those rows; then the user selects X and Y columns to plot (Option A).

The spreadsheet area is editable (cell edit, formulas optional), similar to Excel.

A persistent “AI Insights” button sits at the bottom of the spreadsheet area; clicking it sends the currently visible/filtered dataset to the AI Insights tab where the LLM analyzes and returns structured insights (summary, suggestions, recommended plots, feature extraction).

The Insights output follows a consistent template (example provided) and offers actions (plot, compute, clean) the user can click.

2 — UI / UX details

Layout

[Toolbar: Open/Save | Row selection input | Column select | Plot type dropdown | Plot button]

[Spreadsheet area (editable, selectable cells; supports row-range selection)]

[Plot preview area (right or bottom, depends on layout)]

[Bottom bar inside spreadsheet area: AI Insights button]

Selection interactions

Select columns: click column header (single or multi-select with Ctrl/Cmd).

Select rows (range): click row number for start → Shift+click row number for end OR drag across row numbers. Visual highlight of selected rows.

When a row range is selected, UI displays a small badge: Rows 3–10 selected (8 rows).

After selecting rows, user chooses X and Y columns from dropdowns populated with column names (only numeric columns enabled for Y by default; but user can force-select).

Plot button is enabled only when X and at least one Y selected.

Spreadsheet editing features (minimum)

Cell editing (double-click or Enter)

Paste values and paste images (if cell supports attachments)

Copy / Cut / Paste (basic)

Row insert / delete

Column insert / delete

Sort selected rows by column (asc/desc)

Undo / Redo (optional but recommended)

Use a spreadsheet library like Handsontable, AG Grid, or build on top of a lightweight implementation if custom behavior needed.

3 — Plotting behavior (row-range + column selection)

User selects row range (e.g., 3–10). The dataset is trimmed to those rows.

User selects X column and one or more Y columns. (X column can be any numeric or categorical — if categorical, plotting adapts to bar/boxplot.)

User picks plot style (line, scatter, bar, histogram, heatmap if 2D grid).

On Plot, render plot from trimmed rows only.

Provide display options: smoothing, error bars (if columns for std exist), normalization, legend toggles.

Allow multiple series (plot Y1, Y2 from same row-range).

4 — AI Insights workflow

A) When user clicks AI Insights:

If user has a row-range selected → send only that filtered subset.

If no selection → send the entire currently visible table.

Include metadata: column names, data types (numeric/text/date), number of rows, source filename, any user notes.

B) What is sent to LLM

Payload structure:

```
{  
  "sessionId": "...",  
  "datasetName": "filename or sheet name",  
  "rowRange": {"start": 3, "end": 10}, // null if full  
  "columns": [  
    {"name": "Voltage", "type": "numeric", "sample": [-0.8, -0.799, ...]},  
    {"name": "Current", "type": "numeric", "sample": [-3.79e-6, ...]}  
  ],  
  "rowCount": 8,  
  "fullTablePreview": [ ["-0.800", "-3.79E-06"], ["-0.799", "-0.000004"], ... ],  
  "userRequest": "analyze" // or custom query if user typed one  
}
```

C) Insights tab UI

Shows a structured report with sections:

Quick summary (“What your dataset contains”)

Column-by-column summary (range, mean, median, nan count)

Notable patterns / anomalies

Suggested plots (clickable)

Suggested next steps / analyses (clickable actions: compute capacitance, fit model, denoise)

Option: “Ask follow-up” — chat-style input to query deeper

Each suggested action triggers the app to perform a deterministic operation (plot, compute) using the dataset and show results.

5 — LLM prompt design (system + user prompt template)

System prompt (context):

You are a scientific data analyst assistant. The user uploads experimental or simulation CSV/Excel data. Produce concise, accurate, technical but readable summaries and actionable next steps. Always preserve numeric correctness and suggest plot types and calculations relevant to electrochemistry and physical measurements where applicable. If uncertain, state the assumption.

User prompt (example filled):

Here is a dataset (name: ideal rectangular cv 3 (Sheet1), rows 3–10):

Columns: Voltage (V) — numeric, range [-0.8, 0.8], Current (A) — numeric, microampere scale.

Provide:

A one-paragraph quick summary describing what this dataset likely represents.

A per-column summary (range, typical units, count, missing values).

Notable features or anomalies.

Suggested plots (with reasons) and which columns to use as X/Y.

Suggested numerical analyses (e.g., compute capacitance, calculate mean current, baseline correction) and the formula or method.

Next research suggestions relating this dataset to electrochemical capacitors (if applicable).

Provide results in structured Markdown with headings.

6 — Example output template (LLM result)

Use this template. The app should parse and render it with actionable buttons.

What Your Dataset Contains

Short description paragraph.

Columns (summary)

Voltage (V) — numeric — range $-0.800 \rightarrow +0.800$ — units: V — example values: ...

Current (A) — numeric — range ... — units: A

Key observations

Bullet list of observations, anomalies, signs.

Suggested plots

CV curve — X: Voltage, Y: Current — reason: shows rectangular shape for capacitive behavior — [Plot] button

Current histogram — Y: Current — reason: distribution of baseline — [Plot] button

Suggested analyses

Calculate areal capacitance using formula $C = I / (dV/dt)$ — [Compute] button

Baseline subtraction steps — [Run] button

Next steps / Suggested papers

Short list of recommended follow-ups

If internet search enabled, include 2–3 relevant recent papers with short abstracts (see integration notes below).

Ask follow-up

A chat input for follow-up queries.

7 — Internet search integration (for recommended papers)

If enabled, the Insights flow can optionally call a papers search service (e.g., CrossRef, Semantic Scholar, Google Scholar API wrappers, Bing Academic) to return similar papers and abstracts.

Provide toggle: Include literature suggestions (on/off).

Returned papers: title, authors, year, abstract snippet, DOI/link.

Display them under “Next steps / Suggested papers” with quick actions: Open, Save to Literature tab, Ask assistant to summarize.

8 — Data model & APIs (suggested)

Data models

Dataset (id, name, columns meta, uploadedBy, sourceFileUrl, createdAt)

AnalysisRequest (id, datasetId, userId, rowRange, columns, requestType, status, resultUrl)

Endpoints

POST /api/visualization/plot — { datasetId, rowRange, xColumn, yColumns, plotType, options } → returns plot image/url or data for client render.

POST /api/visualization/ai-insights — { datasetId, rowRange?, columns?, includePapers? } → triggers LLM analysis, possibly internet search. Returns structured JSON (matches template).

GET /api/visualization/dataset/:id/preview — return table preview and column meta.

9 — Frontend components (React)

/components/Visualization

Toolbar.jsx // Row-range input, column selectors, plot type, plot button

Spreadsheet.jsx // editable grid with selection API

RowSelector.jsx // UI helper to select row ranges

PlotArea.jsx // renders plot(s) from returned data

AllInsightsButton.jsx // bottom bar button

InsightsTab.jsx // shows LLM results + actions

Spreadsheet must expose programmatic API: `getSelectedRows()`, `getSelectedColumns()`, `getTrimmedDataset(startRow,endRow)`.

InsightsTab receives structured JSON and renders sections with action buttons that call plotting/computation endpoints.

10 — UX rules & edge cases

UX rules

Show a confirmation modal if user tries to send extremely large datasets (e.g., >100k rows) to LLM; suggest sampling or sending summary statistics only.

While AI Insights is processing, show spinner + progress and allow cancellation.

If network/LLM fails, provide clear error and option to retry.

For plots, keep original dataset intact; plotting uses a copy or transient filtered view.

Edge cases

Non-numeric X chosen → show warning and convert to categorical plot if appropriate (bar/boxplot).

Missing values in numeric columns → LLM should be told about NaN counts; plotting functions should handle gap rendering or imputation options.

Very small or very large magnitudes → auto-detect units (e.g., micro vs amperes) and display scaled axis labels.

11 — Tests to add

Unit tests for `getTrimmedDataset` with various row ranges.

Integration tests for plot endpoint (small dataset).

E2E test for selecting rows 3–10 → choose X/Y → plot → verify plot renders expected number of points.

Tests for AI Insights response parsing and rendering.

12 — Security & performance notes

Sanitize dataset inputs; avoid sending raw PII to LLMs.

Rate-limit LLM calls per user and debounce repeated clicks.

For large datasets, pre-calc summaries (min/max/mean) on server and send summaries instead of full data unless user confirms.

Cache common Insights results for same dataset+rowRange to reduce cost.

13 — Example implementation snippet (client-side flow)

User selects rows 3–10 → call `Spreadsheet.getTrimmedDataset(3,10)` → opens X/Y selectors populated with numeric columns → user selects X=Voltage, Y=Current → click Plot → POST `/api/visualization/plot` with payload → render plot.

User clicks AI Insights → assemble payload including preview and metadata → POST `/api/visualization/ai-insights` → receive structured JSON → render `InsightsTab` with action buttons.

14 — Deliverables for Claude / devs

Implement row-range selection UI + `rowRange` API

Spreadsheet editable area with programmatic selection API

Plot endpoint + client plotting integration

AI Insights endpoint with LLM prompt pipeline + internet search toggle

`InsightsTab` rendering + action handlers for plots and computations

Tests + UX polish (progress indicators, sampling warnings)

1. High-level goals

Add download (PNG, PDF, SVG, etc.) and explicit Copy button in Edit tab for any plot.

Copy button copies the plot as PNG to the clipboard and uploads the PNG to backend storage, returning a stable URL.

Pasting a copied plot (Ctrl+V / Cmd+V) into the Notes editor inserts the image (uploaded & persisted) into the note block.

The Insights tab is a general-purpose AI interaction tab that accepts:

Uploaded PDFs

Uploaded plot images (PNG/JPEG/SVG)

Dataset/previews

Previously-generated plots (by file URL)

The assistant can describe, summarize, and analyze uploaded assets and generated plots.

2. UX / Frontend behavior

2.1 Edit Tab — Plot controls

Each plot (chart area) must include:

Download dropdown button with options:

Download PNG

Download PDF

Download SVG (if supported)

Download CSV (if underlying data present)

Copy (Explicit) button (label: Copy)

Click behavior (single click):

Export chart canvas/SVG to an in-memory PNG (client-side).

Copy PNG to system clipboard using Clipboard API (`navigator.clipboard.write`).

Upload the PNG to backend storage via POST `/api/files/upload` (returns `fileUrl`).

Return success to user (toast): “Copied to clipboard and saved to Notes library” with `fileUrl`.

If clipboard copy fails (browser limitation), fallback to step 3 and inform user: “Image saved — please paste manually.”

2.2 Paste into Notes workflow

When user focuses a Notes block (rich text) and pastes (Ctrl+V / Cmd+V) an image:

Detect the image blob in the paste event.

Immediately upload to backend POST `/api/files/upload` (same endpoint used by copy button).

On successful upload, insert an `` inline in the block (rich text editor content).

Mark the image node with metadata `{ fileId, uploadedAt }` so it persists and can be managed later.

If the image was copied from the app (copy button), because it was already uploaded during copy, paste should prefer to insert the uploaded `fileUrl` instead of re-uploading. Use a short-lived client token or clipboard metadata to associate the clipboard image with the final `fileUrl` to avoid duplicate uploads.

2.3 Notifications / UI

Show toast confirmations for Download / Copy / Paste / Upload success/failure.

Allow user to click the pasted image to open a small modal: View, Download, Replace, Delete.

3. API endpoints & storage (backend)

3.1 File upload API

POST /api/files/upload

Purpose: Uploads file (image/pdf) to persistent storage (S3, GCS, or server filesystem). Returns fileUrl and fileId.

Request:

multipart/form-data with file field file

optional fields: { ownerId, source: "plot"|"notes"|"insights", metadata: { chartId?, datasetId? } }

Response (201):

```
{
  "fileId": "uuid",
  "fileUrl": "https://cdn.example.com/uploads/abc123.png",
  "mimeType": "image/png",
  "size": 102400
}
```

Storage notes:

Save in a user-scoped directory for permission control.

Generate public or signed URLs depending on privacy needs.

Store file metadata in DB (File table) with ownerId, createdAt, originalName, mimeType, fileSize, storagePath.

3.2 File metadata & management

DB File model (example)

```
{
  "id": "uuid",
  "ownerId": "uuid",
  "originalName": "plot.png",
  "mimeType": "image/png",
```

```
"size": 102400,  
"storagePath": "s3://bucket/uploads/...",  
"publicUrl": "https://cdn.example.com/...",  
"createdAt": "ISO",  
"source": "plot|notes|insights"  
}
```

3.3 File deletion, replacement

DELETE /api/files/:fileId — delete/soft-delete file.

PUT /api/files/:fileId — replace or update metadata.

4. Clipboard & upload technical details (client)

4.1 Copy (explicit) flow (recommended implementation)

Export plot to Blob (PNG) using chart library API, e.g.:

Plotly: Plotly.toImage(div, { format: 'png' }) → dataURL → blob

Chart.js: canvas.toBlob(...)

Copy to clipboard:

```
const item = new ClipboardItem({ 'image/png': blob })
```

```
await navigator.clipboard.write([item])
```

Upload to backend:

```
const form = new FormData(); form.append('file', blob, 'plot.png'); fetch('/api/files/upload',  
{ method:'POST', body: form })
```

Store upload fileUrl locally and optionally write metadata into clipboard using navigator.clipboard.writeText(JSON.stringify({fileUrl, fileId})) for same-origin read back (limited by restrictions). Alternatively, store mapping in localStorage keyed by a short token placed also into clipboard text.

4.2 Paste handling in Notes

On paste event, check event.clipboardData.items:

If items includes an image item:

Extract blob → if mapping exists (clipboard text contains mapping token), use associated fileUrl and insert directly.

Else, upload blob and insert returned fileUrl.

Insert `` into block.

Important: Some browsers limit reading clipboard images; still attempt with fallback (user can use Notes upload file button).

5. Insights Tab — multi-modal AI interactions

5.1 Purpose

Insights Tab is a generic AI interaction surface that can be opened from various entry points (Visualization → AI Insights, Edit → Copy → Insights, Literature → AI assistant). It should accept direct user uploads (PDFs, PNG/JPEG/SVG plots), previously uploaded app files (via `fileUrl`), and datasets.

5.2 UI behavior

On opening, user sees a chat-like interface with:

File upload area (drag & drop or upload button)

A message composer (text input)

An action toolbar: Analyze PDF, Describe Plot, Summarize Dataset, Find Similar Papers (toggles)

When a user drags/chooses a file:

Upload to `/api/files/upload`

Show the file in conversation as an assistant/user-attached element

Automatically send an LLM analysis request if user selected (or wait for their prompt)

The AI can reference uploaded files by `fileUrl` and must be able to:

Extract text from PDFs (OCR if necessary)

Parse plots (if plot PNG provided, use vision-capable model or run a plot analyzer to extract axes, ticks, and series)

Provide descriptions, captions, or full technical analysis of plots/PDFs

Keep an Upload History list within Insights for quick reference.

5.3 Backend LLM pipeline for files

When the user requests analysis for a file:

If pdf → run a PDF text extractor (`pdf.js` or server-side PDF text extraction). Extract sections, figures, references. Provide summary to LLM.

If image (plot) → run a plot-analysis pipeline:

Option A: use a multimodal LLM that accepts images (if available).

Option B: run server-side image-to-data extraction (PlotDigitizer, custom script) to extract axis labels, ticks, series, approximate numeric points; send extracted data + image to LLM.

Compose a prompt combining extracted content + instructions (see Prompt section below).

Store the analysis result in AnalysisRequest DB table (cache results to avoid recompute).

5.4 Prompt templates (examples)

System:

You are a domain-aware scientific assistant. The user has uploaded a file [fileUrl]. Provide a clear, structured technical description, include: quick summary (1–2 sentences), what the figure/dataset likely represents, key parameters, possible errors or anomalies, and suggested next steps or computations. Use bullet lists and short headings.

User (for a plot image):

The user uploaded a plot image: [fileUrl]. Attempt to identify:

Axis labels and units.

The number of series and general shapes (e.g., rectangular CV, Gaussian peaks).

Any anomalies (sudden jumps, non-smooth behavior).

Provide suggested numeric extractions or recommend replotting using raw data for precision.

Render answer in Markdown with sections: Quick summary, Observations, Suggested analyses, Next steps.

6. Security, privacy & quotas

Authenticate uploads (attach Authorization header) and ensure ownerId is stored.

Files must be accessible only to owner or team scope; generate signed URLs for external sharing.

Rate-limit file uploads and LLM calls to prevent abuse.

Scan file types and limit upload size (e.g., max 50 MB for images, 200 MB for PDFs—configurable).

Sanitize text extracted from PDFs before sending to external LLMs.

7. Edge cases & fallback behaviors

Clipboard API not available / blocked → fall back to uploading image and copying file URL to clipboard text instead of image. Notify user.

Browser refuses to paste image → show instruction: “Use Upload or use Copy button then click ‘Paste from Clipboard’.”

Duplicate upload avoidance → when Copy button uploads, store fileHash mapping. On paste, check for same hash to avoid re-upload.

Large images → compress server-side to reasonable resolution (e.g., max 2048 px width) while preserving readability.

Unsupported image type → reject with friendly error.

8. Tests to write

Unit tests:

POST /api/files/upload with image/pdf returns 201 and correct metadata.

DB file record created with correct ownerId and metadata.

Integration tests:

Copy button flow: exporting canvas → clipboard write (mock) → upload → fileId returned.

Paste into Notes: paste event with image blob results in upload + image inserted into editor.

Insights file processing: pdf upload → extracted text passed to LLM (mock) → response returned.

E2E tests:

User copies a plot → pastes into Notes → reload page → image persists and displays.

User uploads plot to Insights → assistant returns description with correct sections.

9. Implementation checklist for Claude / devs

Add Copy button to plot controls and implement client-side export to PNG.

Implement clipboard copy using Clipboard API with fallback path.

Implement POST /api/files/upload and File DB model; configure S3/GCS backend.

Implement paste handler in Notes editor to detect image blobs and upload them.

Add image metadata support in Notes block model (store fileId).

Build Insights Tab UI that accepts uploads and file URLs, shows upload history.

Implement server-side file processing pipelines: PDF text extractor, plot-to-data extraction.

Implement LLM request handler that accepts multi-modal inputs and returns structured JSON.

Add toasts & UI feedback for success/failure of copy/paste/upload.

Write tests above; ensure security & rate-limiting.