ChatGPT

Designing Scrollio’s AI Project Brain Repository

Introduction

Creating a “project brain” repository means organizing Scrollio’s code and documentation so that AI coding assistants (like Cursor or GitHub Copilot) have rich, structured context at their fingertips. By encoding Scrollio’s vision, architecture, and conventions into the repository, we empower an AI to build features or write tests *consistently* and *correctly*. In practice, this approach (often called **context engineering**) ensures that AI helpers understand the project’s intent and patterns, reducing failures and hallucinations [1]. For Scrollio – a micro-learning mobile app for teens, adults, and kids – this means the repository should reinforce its educational mission and technical architecture. For example, Scrollio’s goal of turning passive scrolling into focused learning through short, curated videos (with safety features for kids) should be clearly documented as a guiding principle [2] [3]. In short, we need a comprehensive repository blueprint that serves as a single source of truth for both humans and AI.

Why invest in a context-rich repo? Studies show that **missing context is a major cause of AI coding errors** – nearly half of quality issues stem from context gaps [4]. By contrast, supplying thorough project context can boost consistency and enable more complex, multi-step features [1]. In the following sections, we outline how to structure the Scrollio repository, what files to include (and their formatting), how to leverage tools for context retrieval, and strategies for versioning – all tailored to Scrollio’s unique mission and tech stack.

Repository Structure for Maximum Context

To maximize useful context, organize the repository into clear sections for **code**, **documentation**, and **AI guidance**. A possible top-level structure might look like this:

```
scrollio-project-brain/
├── .github/
│   └── copilot-instructions.md      # Custom instructions for GitHub Copilot
├── .cursorrules                     # (If using Cursor) AI rules configuration
├── .cursorignore                    # Ignore non-essential files for AI context
├── docs/                            # Project knowledge base (“brain”)
│   ├── project-overview.md          # Scrollio vision, goals, architecture
│   ├── features.md                  # Feature specs and user stories
│   ├── architecture-diagram.png     # System diagram (embedded image)
│   ├── data-schema.md               # Database schema (Supabase tables, ERD)
│   ├── api-spec.md                  # API or integration specs (if any)
│   └── ... (other reference docs as needed)
├── examples/                        # Code examples/patterns for the AI
│   └── (e.g. sample component or util demonstrating best practices)
```

```
├── src/ (or frontend/)             # React Native app source code
│   └── ... (organized by feature or module)
├── backend/                        # Backend code (if any, e.g. cloud
functions)
├── supabase/                       # Supabase config (SQL migration files,
etc.)
├── README.md                       # High-level project README
├── CONTRIBUTING.md                 # Contribution and coding guidelines
├── CHANGELOG.md                    # Version history of features/changes
└── ... (tests/, etc.)
```

The **documentation hub** (here, a `docs/` folder) acts as the project's knowledge base. All crucial context lives here, separate from the code. *Every project needs a central hub for information* – think of this `docs/` directory as Scrollio's "brain" containing design decisions, architectural diagrams, and records of discussions [5]. This ensures the AI (and human developers) always have a shared understanding of Scrollio's goals, status, and history [6].

Special files for AI guidance: We include files specifically to guide AI assistants. For example, a `.cursorrules` file at the repository root can serve as a *"constitution"* for AI collaborators [7]. In this JSON/ Markdown file, we define how the AI should behave: always load certain context files first, follow coding standards, never violate safety rules, etc. (We'll detail the contents in the next section.) Similarly, a `.github/copilot-instructions.md` can provide repository-wide hints to GitHub Copilot with natural language instructions on building and testing the app [8]. By adding these configuration files, we ensure AI tools start with the right assumptions about Scrollio's structure and quality requirements.

Finally, structure the **source code** itself in an intuitive way (group by feature or layer), and consider generating a *directory map* for the AI. For instance, a `docs/directory-structure.md` (or an auto-updating script) can list the folder tree and module purposes, helping the AI quickly locate relevant code [9] [10]. Consistent, descriptive naming in code (e.g. `QuizScreen.tsx`, `ParentalDashboard.tsx`, `useSupabaseAuth.ts`) will further orient the AI. In short, an organized project layout plus an explicit index of that layout will give the assistant maximum leverage to find what it needs.

Essential Files and Documentation to Include

To build rich context, include a variety of documentation files, each serving a specific purpose. Below are key files we recommend and how to format them for clarity:

- **README.md (Project Overview):** The README should provide a quick yet comprehensive snapshot of Scrollio. Start with a **project description** highlighting Scrollio's mission and target users (e.g. "Scrollio is a React Native mobile app that transforms passive social media scrolling into focused micro-learning through 1-minute videos, with a special kids mode for safe, curated content" [2]). Include sections for **Tech Stack** (mentioning React Native, Supabase, Amazon S3, etc.), **Setup/ Installation**, and a brief **Usage guide**. Also summarize the app's primary features and architecture at a high level. Use clear Markdown headings for each section. For example:

```
# Scrollio – Micro-Learning App
**Mission:** Turn passive scrolling into learning. Scrollio offers a
distraction-free feed of < 1 minute educational videos, curated by AI and
experts, plus quizzes and gamification to reinforce learning [2] . A special Kids
mode provides a safe, monitored learning environment [3] .

## Tech Stack
- **Frontend:** React Native (Expo)
- **Backend:** Supabase (PostgreSQL DB + Auth)
- **Storage:** Amazon S3 (video content)

## Features
- Personalized video feed with AI curation
- Interactive quizzes and progress tracking
- Child-friendly mode with parental dashboard ...

## Installation & Running
(Instructions...)

## Architecture Overview
(Brief intro, with link to docs/architecture.md for details)
```

Formatting tip: Keep paragraphs short and use bullet points for lists of features or tech components. This makes it easy for an AI to parse key facts. Include links to deeper docs (like `[Architecture](docs/architecture.md)`) so the AI can follow them.

- **Architecture & System Design (`docs/project-overview.md` or `architecture.md`):** This is a detailed narrative of Scrollio's design. It should describe how the **frontend**, **backend**, and **storage** interact. For example, explain that the React Native app uses Supabase for authentication and as a database (outlining key tables like Users, Videos, Progress), and uses S3 (or Supabase Storage) for serving video files. An architectural diagram (included as `architecture-diagram.png` and embedded in the Markdown) can illustrate the data flow: user opens the app → fetches personalized video list from Supabase (which contains metadata and S3 URLs) → streams videos from S3 → posts quiz results back to Supabase, etc. Use subsections for each major component:

- *Frontend (React Native):* Describe navigation structure (e.g. stack or tab navigation), state management, and any important libraries (for instance, "video player uses Expo AV" or similar if applicable). Note patterns like functional components with hooks, etc.

- *Backend (Supabase):* Document how Supabase is used – e.g. "Supabase provides a Postgres database and auth; the app interacts via Supabase JS SDK. Key database tables: `profiles`, `videos`, `progress`, ...". If you have any custom Supabase Edge Functions or SQL triggers for business logic, list them here or in a sub-file.
- *Storage (S3):* Explain how video uploads and retrieval are handled. For example, "Videos are stored in Amazon S3. The app obtains a presigned URL from [our backend/Supabase function] to upload or stream videos securely." Include any assumptions (like video size limits or formats).

Also mention cross-cutting concerns: **security & safety** (e.g. data validation, content moderation for kids), and how the architecture enforces them. For instance, note that certain collections in the feed are curated by human moderators (to ensure quality for topics like finance) [11], or that the Kids mode feed is restricted to pre-approved content. By spelling out these design intents, an AI will be less likely to introduce code that violates them. *Formatting:* use diagrams, bullet lists for key points, and keep a logical flow from high-level components to finer details. This document essentially teaches the AI "how all the parts of Scrollio fit together" before it writes or changes any code.

- **Feature Specifications (`docs/features.md` or individual feature docs):** Provide context on Scrollio's major features and user stories. This can be one consolidated **Features** document with subheadings per feature, or separate files (e.g. `docs/feature_feed.md`, `docs/feature_quiz.md`, `docs/feature_kids_mode.md`). Each feature section should outline:
- **User Story & Purpose:** e.g. "*Feed:* The user scrolls through a vertical feed of 60-second educational videos tailored to their interests. Solves the need for quick learning without distraction [2]."
- **Key Components:** list the app screens, components, or modules involved. For instance, the feed might involve `FeedScreen` component, a `VideoCard` sub-component, and a context/provider for the feed data.
- **Workflow:** describe how the feature works step-by-step. E.g., "On app launch, the app fetches `videos` from Supabase where `topic` matches the user's preferences. Videos are displayed one-by-one. When the user reaches certain intervals, a quiz modal pops up (see Quiz feature)."
- **Design Constraints or Rules:** any important guidelines for that feature. For example, for *Kids Mode*, note that only content with a certain flag or from a curated list is shown, and usage time might be limited (to align with parental control goals). This alerts the AI not to bypass these rules. For the *Quiz* feature, mention that quiz questions should map to video content and how scoring is recorded in the database. For *Drawing-to-Narrator feature*, describe the pipeline (child draws a shape → an AI service transforms it into a 3D avatar; this might involve an external API or library – document any integration details if relevant).

Use bullet points or numbered steps for workflows to keep it scannable. Where possible, link these descriptions to code once it exists (e.g., "(See `src/components/VideoCard.tsx`)"). The feature specs help an AI understand the intent behind code sections, ensuring new code aligns with intended behavior.

- **Data Schema and API Docs (`docs/data-schema.md`, `schema.sql`, `api-spec.md`):** It's critical for the AI to know the data structures it's dealing with. If you use Supabase, *export or document the database schema*. This could be a Markdown table listing each table, its columns, and their meaning, or an ERD diagram. For example:
- **Users table:** `id (uuid, primary key)`, `email`, `name`, `role (learner/parent/child)`, etc.
- **Videos table:** fields for video metadata (id, title, topic, URL to S3, duration, age_category etc.).
- **Progress/QuizResults table:** how you track points or quiz scores per user.

If the schema is managed via SQL migration files (common with Supabase), include those in a `supabase/migrations/` folder so the AI can see the source of truth. Providing the schema lets the AI correctly write queries, avoid introducing nonexistent fields, and maintain compatibility with the database. It can simply look up how "progress" is stored rather than guessing.

Similarly, if the app communicates with any external APIs or has backend functions, document their interfaces in an **API spec** file. For instance, if there's a cloud function for generating the presigned S3 URL, document its endpoint and usage (input: file name, output: URL). This prevents the AI from inventing incorrect API calls. Use OpenAPI format if you have many endpoints, or a simple list of endpoints with method, path, and purpose if only a few.

- **Coding Guidelines and Conventions (`CONTRIBUTING.md` or `docs/contributing.md`):** This file guides human contributors and AI on how to write code that fits Scrollio's style. Outline the preferred **code style** (for React Native, e.g. functional components with hooks, using TypeScript if applicable or PropTypes, naming conventions for files and variables), and the **branching/commit strategy** if relevant. For example, specify: "Follow [Airbnb's JavaScript style guide] for React components, use 2 spaces for indentation, and descriptive names. Commit messages should be in imperative mood and mention the feature or fix. All new features must come with unit tests."

In particular, include any **patterns or frameworks** the project uses: e.g., "State management via React Context API for user auth state", or "UI components follow a design system (colors, fonts specified in `theme.js`)". If Scrollio has both adult and kid modes, note how that is structured (maybe a flag in user profile and conditional rendering – document this so the AI doesn't hardcode values incorrectly). By formalizing these conventions, we ensure the AI writes code in the same style and avoids common mistakes. *Format tip:* Use bullet points for each rule or convention, and separate sections for different areas (coding style, testing, git workflow, UI/UX guidelines, etc.). This is essentially a condensed checklist the AI can refer to during coding.

- **AI Guidance & Prompting Rules:** Perhaps the most important part of an AI-first repository are the files that directly instruct the AI. As mentioned, `.cursorrules` (for Cursor) or `CLAUDE.md` **/ custom instructions files** serve this role. In these, write out rules in plain language (or JSON if the tool expects it) covering:
- *Project context to load:* e.g. "Always read `docs/project-overview.md` and `docs/task-list.md` at the start of each session" [12]. This ensures the AI refreshes itself on Scrollio's context every time it's about to generate code.
- *Operational protocols:* define how the AI should approach coding tasks. For example: "Before writing code for a new feature, break the task into subtasks (e.g. update database schema, create new React component, write tests) and confirm the plan" [13]. And: "When modifying existing code, **always** read the relevant file or module first to understand context [14], and preserve existing functionality." These rules prevent the AI from jumping in blind and possibly breaking things.
- *Safety and quality requirements:* list non-negotiable rules like "Never hardcode secrets or credentials", "All user input must be validated (to maintain parental control safety)", "Do not introduce content that isn't curated for kids" etc. For instance, reinforce Scrollio's safety goal: *if generating any content or suggestions for kids mode, it must adhere to curated content rules* [3]. Also include general coding safety: "Do not disable TypeScript checks" (if TS is used), "Handle errors by logging and using our error UI component", etc. [15].
- *Testing and documentation:* e.g. "Whenever you add new code, also generate corresponding unit tests in the `__tests__` folder, and update documentation if applicable". This reminds the AI to keep the project well-tested and docs up-to-date.
- *Project-specific preferences:* If there are any particular frameworks or libraries to prefer (or avoid), list them. For example, "Use Supabase JS client for all database operations (do not use raw `fetch` calls

to Supabase REST endpoints)", or "Prefer using the provided `VideoPlayer` component for playing videos instead of creating a new one."

These instructions can be maintained as a simple Markdown checklist or structured rules. GitHub Copilot's `copilot-instructions.md` can be very straightforward text ("This project uses X, do Y, don't do Z"), whereas Cursor's `.cursorrules` might be JSON with sections as in the example above. An example from a practical guide defines rules ensuring the AI always reads the overview first, follows certain coding protocols, and never breaks type safety [16] [17]. By establishing such ground rules, you put the AI "on the same page" as the development team regarding priorities and standards [18].

Formatting: Make this file as explicit as possible – it's essentially your proxy for engineering judgment. Use **ALWAYS/NEVER** phrasing for critical rules (as shown in Cursor's example) and include rationale comments if needed. Keep it updated as the project evolves.

- **Example Code Patterns (`examples/` directory):** Including a few minimal, well-commented code examples can massively help an AI understand how you want things done. These could be real snippets from the project or simplified stand-alone examples. For instance, you might add:
- `examples/database_usage.ts`: showing how to initialize the Supabase client and perform a query or insert, following the project's error handling patterns.
- `examples/video_component.tsx`: a tiny React Native component illustrating best practices (proper use of hooks, styling with StyleSheet, etc.).
- `examples/api_call.ts`: if the app calls external APIs or cloud functions, a template of how to do that with proper error checks and loading states.

These serve as reference implementations. In your prompting workflow, you can even instruct the AI to refer to specific example files when implementing similar functionality. For GitHub Copilot or others that automatically ingest the repo, just having these files present provides additional training signals. Make sure the examples are **representative and up-to-date** – they should reflect the current preferred patterns. Each example file should have a top comment explaining its purpose (for the human reader) and perhaps a note in documentation referencing them ("see `examples/` folder for usage patterns"). As the context-engineering template notes, example files are *critical* for showing the AI exactly how to follow your project's patterns [19].

- **Testing Documents:** While not always required, it could be useful to have a short `docs/testing.md` that outlines how the project approaches testing. For Scrollio, this might state which testing framework is used (e.g. Jest for React Native), how to run tests, and any testing conventions (like using React Native Testing Library for components, or writing end-to-end tests for critical user flows). If certain features have specific test requirements (e.g. video playback or quiz logic might need special handling or mocking), mention that. This context will encourage the AI to produce test code that fits in. Additionally, you can maintain a "test checklist" that the AI can consult to ensure a feature is properly covered (for example: "For any new component, ensure there is a test rendering it with sample props, a test for each major interaction like button presses or video end event, etc.").

In all these files, **formatting and clarity are key**. Use Markdown headings (`##`, `###`) to delineate sections, and lists (`-` or `1.`) for clarity where appropriate. The content should be easily skimmable – an AI might not interpret long-winded paragraphs as reliably, but bullet points and concise sentences it can digest. By combining narrative sections (for the big picture) with structured lists and code examples (for specifics), we cater to the AI's need for both broad context and concrete references.

Enhancing Context Retrieval with Semantic Search and Tools

Even with a well-structured repository, an AI assistant may not automatically “know” which file holds the answer to its questions. This is where augmenting the AI with **retrieval tools** comes in. We recommend integrating technologies like **vector databases**, semantic search, or OpenAI’s function-calling mechanisms to help the AI fetch relevant context on demand.

1. Build a Semantic Index of the Repository: Use embeddings to represent the meaning of your documentation and code, enabling semantic searches. For example, you can leverage Supabase’s built-in pgvector support or an external vector DB to store embeddings of all `docs/*.md` content (and possibly important code files). Each document or section is turned into a vector so that later, given a query (like “how does authentication work?”), you can retrieve the most relevant snippets by cosine similarity. Supabase provides tools to do this efficiently [20] [21], or you could use open-source solutions (e.g. FAISS, Pinecone). The **embedding index** essentially becomes the AI’s extended memory: rather than stuffing all docs into the prompt, the AI can query the index for what it needs.

2. Use LangChain or Similar Frameworks: LangChain is a popular framework for chaining LLMs with tools and data [22]. You can set up a **Retrieval-Augmented Generation (RAG)** pipeline specifically for Scrollio’s repo. For instance: - **Document Loader:** A script (Python or Node) that reads all Markdown files (and perhaps code files) and uses an embedding model (OpenAI’s text-embedding, etc.) to embed them. Store these in a vector store. - **Retriever Function:** At runtime, given an AI query or feature request, use the vector store to **search semantically**. This finds relevant pieces of text even if specific keywords differ (e.g. a search for “video list” might match “feed” or “videos table schema”). - **AI Agent Integration:** Incorporate this retrieval as a tool the AI can call. For example, if using OpenAI’s function calling, you might define a function `search_docs(query: string) -> string` that when invoked by the AI, will perform the vector search and return the top relevant doc excerpt. The AI’s prompt can then include those excerpts as additional context before it formulates code. This way, even if the repository grows large, the AI can always pull in the most pertinent 2-3 pieces of context for the task at hand.

Practical example: Suppose the AI is tasked with adding a new quiz type. The agent can first call `search_docs("quiz format")` which might return a snippet from `features.md` about quizzes and a snippet from `data-schema.md` about the QuizResults table. The AI reads those, and then proceeds to generate code knowing how quizzes are supposed to work in Scrollio. This approach was illustrated in an AI coding workflow where the system would *“analyze the codebase for patterns”* and *“search for relevant documentation”* before implementation [23] – effectively mimicking how a human would consult design docs and existing code.

LangChain can also be used to **summarize code files** if needed. For instance, you might periodically run a script to summarize each source file’s purpose (using an LLM) and store those summaries in a `docs/code-summaries.md` or in the vector DB. This gives the AI a high-level view of each module without reading all code. One engineer described generating per-file summaries and then merging them into a full repository summary [24] [25], which can be extremely useful for onboarding an AI to the project. For Scrollio, summarizing key files (like the main navigation setup, the video player component, etc.) can help the AI quickly orient itself within the code structure.

3. Utilize Function Calling and Tools in the AI Environment: Modern AI assistants allow defining custom tools. Beyond just documentation retrieval, consider tools like: - **Code Search:** A function that does a keyword search within the repository (e.g., regex or full-text). This is handy if the AI wants to find where a function is defined or where a variable is used. Copilot in VSCode, for example, can reference the workspace files with `#` mentions [26], and Cursor has features for multi-file selection [27]. But for autonomous runs, a `find_in_repo("Keyword")` tool could be defined. - **Run/Test Commands:** If your setup permits, allow the AI to run the project's tests or a development server in a sandbox. For instance, an `npm_test()` function could execute `npm run test` and return results. This way the AI can verify that its changes didn't break anything and even iterate (fix failing tests) – a capability some advanced AI agent frameworks utilize [28]. This goes beyond repository design, but it's facilitated by having a consistent testing setup documented as above. - **Knowledge Base Q&A:** Another idea is to use a Q&A chain on the docs: the AI could ask a question in natural language (like "What's the difference between Scrollio Standard and Scrollio Kids modes?") and a QA system (LangChain with an LLM) finds the answer from the docs. This essentially gives the AI an interactive way to clarify requirements using the repository content itself as the knowledge base.

By combining a structured repo with semantic search and tools, we achieve a sort of *context on demand*. The AI doesn't need everything in its prompt at all times – instead, it knows **how to retrieve** what it needs. In implementing this, you might maintain a separate script or sidecar application (maybe as part of the repository in a `scripts/` folder) to manage the vector index. For example, a script `scripts/update_index.py` could be run whenever docs are updated to refresh the embeddings in the vector database.

LangChain & Vector DB in action: One could integrate this with the development workflow by using an agent that orchestrates tasks. Imagine a custom CLI command `ai_feature "Add profile picture upload"` that triggers the agent. The agent would use the repo context: it might call search functions, read the `project-overview.md` for any mention of user profiles, and then plan code changes. The blueprint repository we're designing fully supports this by having all the key info (profile data model, how file storage works, etc.) readily available to be fetched. In essence, the repository + retrieval tools turn AI into an autonomous or semi-autonomous contributor that "knows" Scrollio almost as well as a team member.

Versioning, Naming, and Long-Term Maintenance Strategies

Setting up the repository is not a one-and-done task – maintaining context quality over time is crucial. Here we outline strategies for versioning documentation, naming conventions, and keeping context coherent as Scrollio evolves:

- **Documentation Versioning:** As Scrollio's features and architecture change, periodically update the docs and consider using a **CHANGELOG** or **release tagging** for context. A `CHANGELOG.md` can record each major feature addition or change (with dates and version numbers). Not only does this inform developers, but an AI agent could read the changelog to understand what's new or what breaking changes occurred in each version. For example, if version 2.0 redesigns the onboarding flow, the changelog notes can alert the AI that any old code or docs about onboarding might be outdated. In cases of significant pivots, you might keep older documentation in versioned folders (e.g., `docs/archives/v1_architecture.md` for historical reference), but generally it's better to keep only the latest truth in main docs to avoid confusion. Tools like the context template versioning example [29] show how teams can maintain different sets of context for v1.0 vs v1.1, along with a changelog – in our case, the repository itself can serve that role via branches or tags if needed.

- **Consistent Naming Conventions:** Establish clear naming schemes for files, directories, and identifiers, and document these rules. Consistency in naming aids context coherence – the AI can infer the purpose of files from their names. For instance:

- **File/Folder names:** Use self-descriptive names like `FeedScreen.tsx`, `QuizModal.tsx`, `ParentDashboard/` directory, etc. Avoid ambiguous names. If using domain terms, match them to the PRD nomenclature (e.g., stick with "Learner" vs "Standard" user consistently across docs and code).
- **Git Branches and Versions:** If collaborating via Git, consider a naming scheme for branches (feature branches named `feature/quiz-results`, etc.) and tag releases (e.g., `v1.0-beta`). This doesn't directly feed the AI in real-time, but it helps organize development for humans, which in turn keeps the repository cleaner for the AI.

- **Code identifiers:** Follow a style guide so that components, classes, and functions have meaningful names. If the design calls certain concepts "mentor" (AI narrator) or "playground" (kids 3D world), use those words in the code element names rather than something obscure. Keeping terminology consistent between the PRD, docs, and code will prevent the AI from misinterpreting (for example, it will clearly tie the concept of "mentor" in a user story to the `MentorCharacter` class in code if naming aligns).

- **Documentation Updates as Part of Workflow:** Encourage a practice that **every code change that affects behavior or architecture should come with a doc update**. If the AI is making changes, we can even encode this requirement in its instructions (e.g., in the AI guidelines: "After implementing a new feature, update the relevant section in `docs/features.md` and add an entry to CHANGELOG"). This keeps the context current. Stale documentation can mislead the AI in the future, so treating docs as living documents is vital. In pull request templates (if using GitHub), you might add a checkbox like "Docs updated?". Some AI systems can be instructed to refuse merging code that doesn't include necessary doc changes (through a CI check or through the AI's own validation step).

- **Use of ADRs (Architecture Decision Records):** For any major architectural or technical decision, consider adding a short ADR markdown file (e.g., `docs/adrs/2025-12-03-auth-method.md` explaining why you chose Supabase Auth over a custom solution). These are useful for human onboarding, but also for AI: it provides reasoning context. For example, if later an AI suggests "let's switch authentication to a different service," an ADR file might explain constraints that led to Supabase – the AI seeing that might reconsider or at least inform its suggestion. ADRs should be succinct (problem, decision, reason format) and can be indexed by the retrieval system as well. Over time, they create a narrative of the project's evolution which aids complex reasoning.

- **Monitoring and Improving AI Interactions:** As the AI begins contributing, use version control to your advantage to track its performance. Tag AI-generated commits with a marker (maybe include `AI:` in commit messages) to later analyze how often AI contributions need follow-up fixes. If you notice certain kinds of mistakes recurring, update the repository context or rules to address that. For example, if the AI often forgets to update a particular config file, add a note about that in the guidelines. The repository is not static – it should be continually refined to better steer the AI. Teams using such context-engineering report doing *"weekly context reviews"* to discuss what improvements can be made [30]; you can adopt a lightweight version of that by periodically reviewing if the docs are accurate and if the AI is misunderstanding anything.

- **Long-Term Context Coherence:** As Scrollio grows, you might have more and more documentation. Organize the docs folder logically (you can subdivide by topics if it gets large, e.g., `docs/frontend/`, `docs/backend/`, `docs/product/` etc.). Maintain a **Table of Contents** in the main README or in `docs/README.md` that lists all available documentation resources. This serves as a guide for contributors and a directory for the AI. Moreover, if some features are removed or radically changed, don't be afraid to prune or archive the old context. For instance, if the "drawing interface for mentor" feature is postponed or deprecated, you should mark that in the documentation (or move that section to an archive). An AI reading the docs should clearly see what's current versus legacy. You can use call-outs or badges in docs (e.g., "**[Deprecated]** feature, not in current version") to signal that. This prevents confusion where the AI might otherwise try to support a removed feature.

- **Example of Naming/Doc Consistency:** Let's say Scrollio introduces a premium tier for extra content. You'd add this to features documentation, name related code as `PremiumContentService` or `PremiumFlag`, update the database schema (new field `is_premium` on users perhaps) *and* document that. If six months later an AI is asked to add a feature dependent on premium status, all the relevant pieces are in the context for it to discover. The AI might search the vector store for "premium" and find the schema entry and feature description instantly. This cohesive linkage across naming in docs and code is why consistency is emphasized – it's like leaving a trail of breadcrumbs for the AI to follow.

Conclusion

By implementing the above blueprint, the Scrollio repository becomes a rich, self-contained knowledge hub that an AI agent can use to effectively contribute to the project. We have structured the repository to include every piece of context an AI (or new human developer) would need: from high-level vision down to code patterns. We've also integrated modern AI tooling concepts – semantic search, vector databases, and clear function interfaces – so the AI can retrieve the right information at the right time.

This approach is **actionable**: one can start by creating the recommended docs (using content from the Scrollio PRD for the vision and features) and setting up the special AI instruction files. Populate the knowledge base (`docs/` and examples) with the current understanding of the system. As development proceeds (whether by humans, AI, or both), enforce updates to this context with each change. The result will be a virtuous cycle: a well-documented project that **accelerates AI development**, and AI development that remains aligned with Scrollio's mission of education and safe learning.

With this project-brain repository in place, an AI coding assistant can autonomously generate new features, write tests, or refactor code while **staying true to Scrollio's architecture and intent**. In essence, we are encoding the team's collective wisdom and the product's soul into the repo. This not only boosts the AI's effectiveness but also ensures the longevity and maintainability of the app as it grows. It sets Scrollio up for success, allowing the developers to leverage AI as a reliable partner in building an innovative learning platform.

Sources:

- Scrollio Product Requirements Document [3] [2] [31]
- Context Engineering Template (Coleman's GitHub) [32] [33]
- "Mastering Long Codebases with Cursor" – AI project setup guide [16] [5]
- GitHub Copilot documentation on repository instructions [8]
- Upsun Blog on Context Engineering benefits [34]
- TowardsAI article on AI code explainer (LangChain usage) [24]
- Cursor Community Forum discussions on context files and project brain setup [35] [12]

---

[1] [19] [23] [28] [32] [33] GitHub - coleam00/context-engineering-intro: Context engineering is the new vibe coding - it's the way to actually make AI coding assistants work. Claude Code is the best for this so that's what this repo is centered around, but you can apply this strategy with any AI coding assistant!
https://github.com/coleam00/context-engineering-intro

[2] [3] [11] [31] Final Product Requirements Document (PRD)_ Scrollio.docx
file://file_00000000af5c71f488228d342e69c7a4

[4] [29] [30] [34] Context engineering for AI-assisted development: why it matters
https://upsun.com/blog/context-engineering-ai-web-development/

[5] [6] [7] [9] [10] [12] [13] [14] [15] [16] [17] [18] [35] Mastering Long Codebases with Cursor, Gemini, and Claude: A Practical Guide - How To - Cursor - Community Forum
https://forum.cursor.com/t/mastering-long-codebases-with-cursor-gemini-and-claude-a-practical-guide/38240

[8] Adding repository custom instructions for GitHub Copilot - GitHub Docs
https://docs.github.com/en/copilot/how-tos/configure-custom-instructions/add-repository-instructions

[20] AI & Vectors | Supabase Docs
https://supabase.com/docs/guides/ai

[21] How do I use supabase as knowledge? - Questions - n8n Community
https://community.n8n.io/t/how-do-i-use-supabase-as-knowledge/160388

[22] [24] [25] I Built an AI That Reads Any GitHub Repo and Explains the Code (Using LangChain) | by Sanjay Nelagadde | Nov, 2025 | Towards AI
https://pub.towardsai.net/i-built-an-ai-that-reads-my-github-repo-and-explains-the-code-using-langchain-a0c2e34c4c07?gi=897d4e456892

[26] Manage context for AI - Visual Studio Code
https://code.visualstudio.com/docs/copilot/chat/copilot-chat-context

[27] Enhanced Repository Context and Multi-File Selection
https://forum.cursor.com/t/enhanced-repository-context-and-multi-file-selection/36939