Design Document

Crowdsourcing Code Comments with ChatGPT (CrowdDoc)
CSE 4101 Senior Project - Fall 2026
Ahmed Alzahrani and Mohammed Zakry
Faculty Advisor / Client: Professor Marius Silaghi

1. Design Overview

CrowdDoc uses a simple web architecture. A user works in a normal browser. The browser talks to a Next.js application server. The server loads the saved C source, parses functions, reads or writes explanations and votes, and can call the Writer and Checker / Reviewer AI.

For Milestone 1, the source is a small Linux v6.12 snapshot with three C files and eleven selected functions. Keeping the source small makes it easier to test the full idea before adding a larger repository.

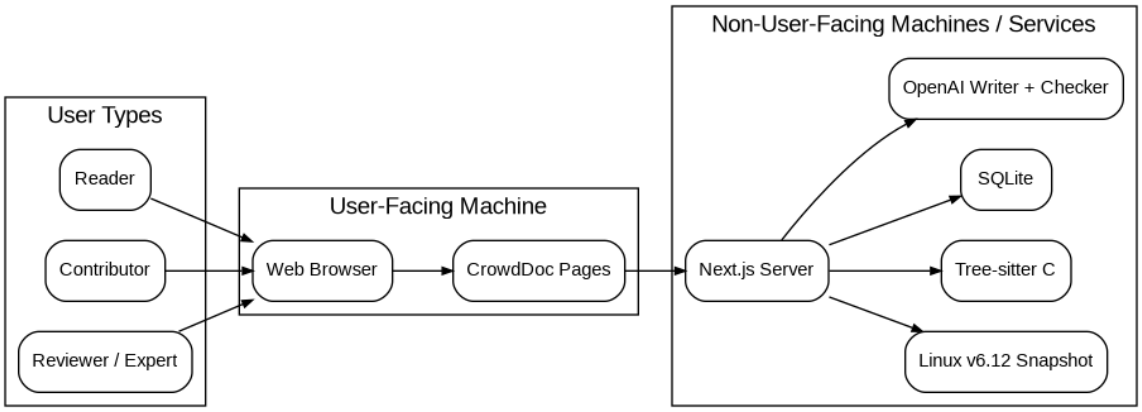

Figure 1. High-level architecture using the three required areas.

In the current website, the second AI agent appears as Reviewer. This document calls it Checker / Reviewer to match both the project idea and the website label.

2. Main Components

Web Browser / User Interface: shows folders, files, function source, explanations, forms, and vote controls.

Application Server: receives requests, checks input, loads the selected function, starts AI calls, and reads or writes the database.

Source Browser: reads the saved Linux v6.12 snapshot and lists known folders/files.

C Parser: uses Tree-sitter C to find supported function definitions and basic function information.

AI Module: runs the Writer first and then runs the Checker / Reviewer using the saved Writer result.

Database: SQLite stores function records, explanations, and votes without changing the Linux source files.

3. Module Interfaces / Simple UML

The project uses TypeScript modules and functions instead of large Java-style classes. The UML below is a simple view of the main modules and methods that are already in the project.

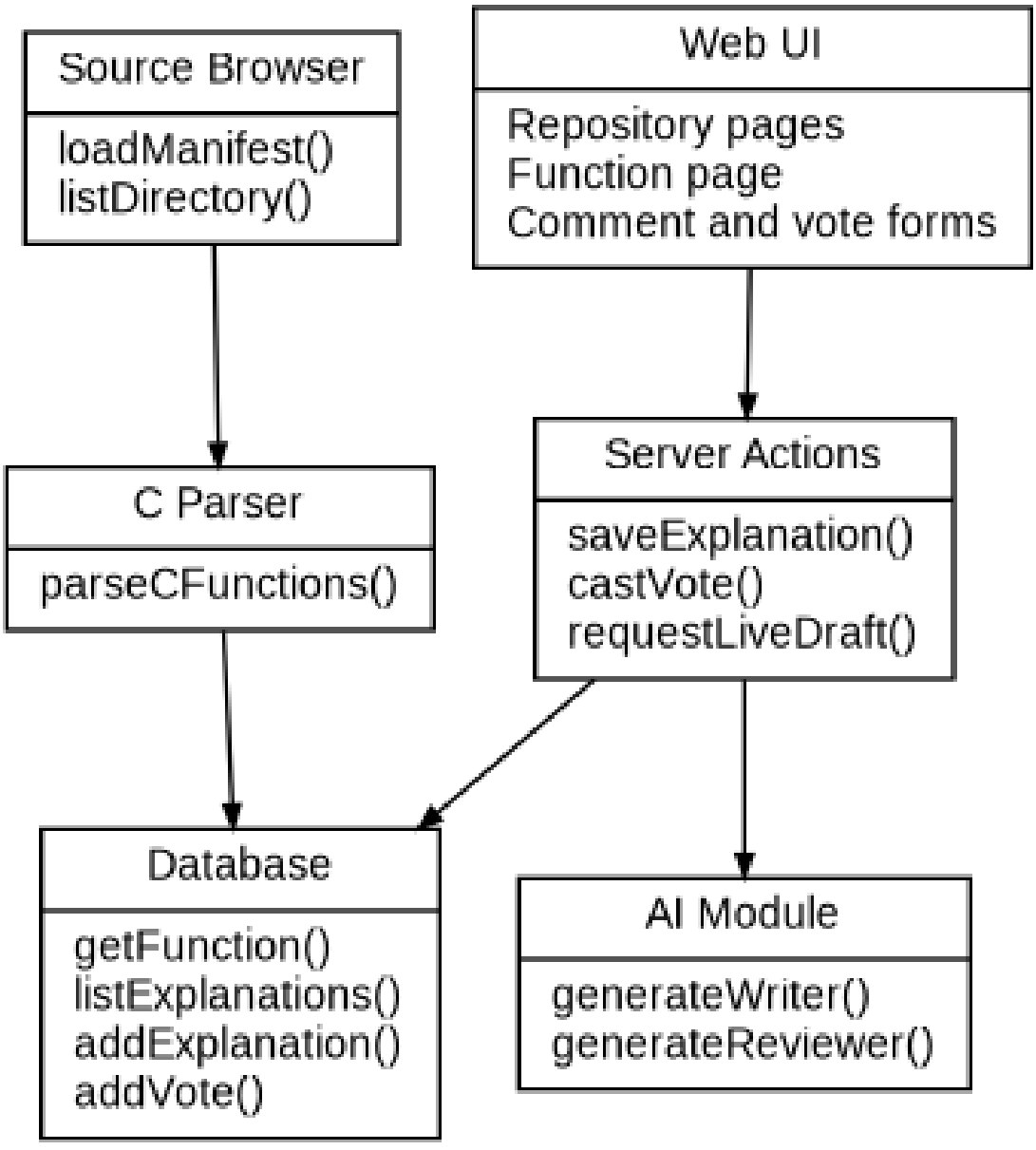

Figure 2. Simple UML-style module diagram.

4. Module Details

4.1 Source Browser

loadManifest() reads the source manifest. listDirectory() returns the known folders and files. Path checks reject unsafe values such as '..'.

4.2 C Parser

parseCFunctions() uses Tree-sitter C to return the function name, source, signature, line range, and content hash.

4.3 AI Module

generateWriter() creates the first explanation. generateReviewer() is the Checker / Reviewer step and receives the same function plus the Writer explanation.

4.4 Database Module

getFunction() loads one function. listExplanations() loads its comments. addExplanation() saves AI or human explanations. addVote() saves a vote.

4.5 Server Actions

saveExplanation() validates and saves a human correction/addition. castVote() checks the relationship before saving a vote. requestLiveDraft() runs the live Writer and Checker / Reviewer flow.

4.6 Web Interface

Next.js pages and React components show the source browser, function page, explanations, human form, breadcrumbs, and voting controls.

5. Main User Flow

1. The user opens CrowdDoc.
2. The user browses the saved Linux v6.12 folders and files.
3. The user opens one of the selected C functions.
4. CrowdDoc shows the exact read-only source.
5. The Writer explanation is shown or a live Writer call is requested.
6. The Writer result is saved.
7. The Checker / Reviewer receives the same function and saved Writer result.
8. The Checker / Reviewer / Reviewer result is saved and shown.
9. A user can add a human correction or addition.
10. A user can vote for an explanation.

6. User Interface Mock-up

The interface is kept simple. The user first browses the source, then opens a function page, and can add a human correction from that page.

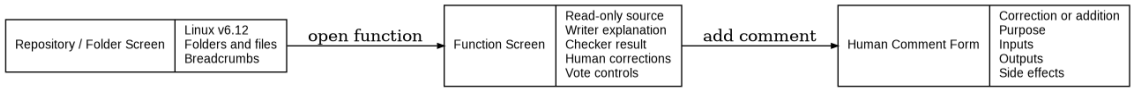

Figure 3. Simple GUI mock-up based on the current website flow.

7. Database Design

A Function record represents one exact function version. An Explanation belongs to one Function. A Checker / Reviewer result or human correction can point back to the explanation it answers. A Vote belongs to an explanation.

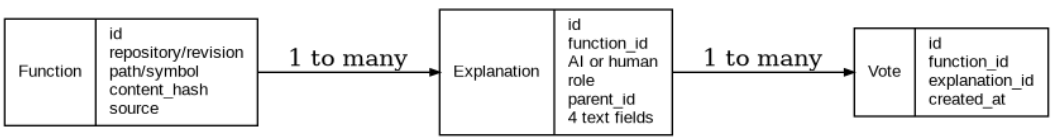

Figure 4. Simple database relationship diagram.

8. Writer and Checker / Reviewer Design

The Writer and Checker / Reviewer are two separate model calls. The Writer receives one C function and creates the first explanation. The Checker / Reviewer receives the same function plus the exact Writer explanation that was saved. This makes the second call a real check of the first result.

If no OpenAI key is configured, prepared Writer and Checker / Reviewer examples can be shown as stand-ins. The page must make that clear. If a live request fails, the user sees a short error message and the API key is not shown.

9. Input Validation and Error Handling

- Reject unsafe source paths such as '..'.
- Require text in all four human explanation fields.
- Limit the current human explanation fields to 4000 characters each.
- Only allow a human correction to answer an AI explanation from the same function.
- Only save a vote when the function and explanation match.
- Keep the Writer result if the Checker / Reviewer call fails after the Writer succeeds.

10. Communication Between Parts

- Browser to server: normal web requests and Next.js server actions.
- Server to saved source: reads the Linux v6.12 snapshot.
- Server to SQLite: saves and loads functions, explanations, and votes.
- Server to OpenAI: sends the selected function to Writer, then the same function plus saved Writer text to Checker.
- External source link: opens the matching Linux v6.12 file in Elixir/Bootlin.

11. Current Limits and Next Work

The current prototype is intentionally small. It has three Linux v6.12 C files and eleven selected functions. It does not yet import a full arbitrary repository. Sign-in is not part of Milestone 1, and reliable user identity, expert verification, and one-vote-per-user protection are planned next-step items after the basic source/comment flow is stable.