# Structure-Aware RAG Agent for GNU Coreutils

Dhruv Roy Talukdar
*Project Prototype Report*

December 6, 2025

## 1 Problem Statement

The objective is to create a code explanation agent specifically for `coreutils` (written in C). The system must utilize LangChain and a Vector Database. Crucially, the system must support four distinct types of user queries:

1. **Specific Functionality:** "What does the `ls` function do?"
2. **Implementation Details:** "How is memory managed in `xmalloc`?"
3. **Structural Relationships:** "Which functions call `delete_file`?" (Callsite Analysis)
4. **Discovery & Recommendation:** "I want to safely create a temporary file; which functions should I use and how do they help?" (Intent Matching)

To achieve this, the system requires a preprocessing pipeline that separates code from comments and extracts metadata regarding function relationships.

## 2 System Architecture

The solution is divided into three layers: The Ingestion Layer (Parsing), the Storage Layer (Vector DB), and the Application Layer (Agent).

### 2.1 The Ingestion Layer (Structural Parsing)

Standard text splitters are insufficient for C code as they sever context. We will use **Tree-sitter** (specifically `py-tree-sitter`) to parse the C Abstract Syntax Tree (AST).

- **Code/Comment Separation:** The parser will traverse the AST to identify `function_definition` nodes. It will extract the byte-range of the function body for the "Code Chunk" and identify preceding sibling nodes to extract the "Comment Chunk."
- **Callsite Extraction:** Within every function body, the parser will walk the AST to find `call_expression` nodes. This generates a list of "called methods" for every function, which is critical for answering structural questions.

### 2.2 The Storage Layer (Vector Database)

We will use a Vector Database (e.g., ChromaDB) to store two distinct types of documents. This separation allows the LLM to retrieve the most semantically relevant data type based on the user's intent.

| Document Type | Content | Metadata Fields |
|---|---|---|
| **Documentation** | The natural language comment block explaining the function. | `type: "doc"`, `func_name: "ls"`, `file: "ls.c"` |
| **Implementation** | The raw C code of the function body. | `type: "code"`, `func_name: "ls"`, `calls: ["printf", "xmalloc"]`, `file: "ls.c"` |

Table 1: Schema for Vector Storage

### 2.3 The Application Layer (LangChain Agent)

We will implement an agent (e.g., OpenAI Functions Agent) equipped with specific **Tools**. The agent acts as a router, deciding which retrieval strategy to use.

## 3 Proposed Tools for the Agent

Instead of a single "search" function, the agent will have access to specialized tools to handle the four query types defined in Section 1.

1. `lookup_functionality(function_name):`
   - *Use Case:* Explaining a known function.
   - *Mechanism:* Exact match on `func_name` retrieving the document where `type="doc"`.
2. `lookup_implementation(function_name):`
   - *Use Case:* Reading actual code.
   - *Mechanism:* Exact match on `func_name` retrieving the document where `type="code"`.
3. `find_callers(target_function):`
   - *Use Case:* Identifying callsites (dependencies).
   - *Mechanism:* This does *not* use vector similarity. It queries the database metadata: "Return all documents where `target_function` exists in the `calls` list."
4. `discover_relevant_functions(task_description):`
   - *Use Case:* **Discovery & Recommendation** (e.g., "I want to sort files").
   - *Mechanism:*
     (a) Performs a **semantic similarity search** on documents where `type="doc"`.
     (b) Retrieves the **Top-5** results (Top-K retrieval).
     (c) *Agent Synthesis:* The LLM receives these 5 summaries and synthesizes an answer explaining *why* these functions are relevant candidates for the user's goal.

## 4 Implementation Roadmap

### 4.1 The Parser Prototype

**Goal:** Transform raw `.c` files into JSON objects.

- Install `tree-sitter` and `tree-sitter-c`.

- Write Python logic to extract Comments vs. Code and build the Call Graph (List of callsites).
- Validated output: JSON containing code, comments, and call lists.

Configuring vector-database and Agent will trivial.