## Low-Level Design (LLD): Code Reviewer with Llama 3 LLM

### 1. Introduction

1.1 Purpose

This Low-Level Design (LLD) document details the internal logic and code-level design of the "Code Reviewer" application, built using Python, Streamlit, LangChain, and the Llama 3 LLM. This document is intended for developers to understand the implementation of each component and guide the coding process.

1.2 Scope

The LLD will cover the following aspects of the Code Reviewer application:

* Module-level design: Describing the functionality of app.py and helper.py.
* Class design: Outlining the structure and methods of the ResponseLLM class.
* Function-level design: Detailing the logic within key functions.
* Data structures: Specifying data formats and structures used.

### 2. Module Design

2.1 app.py (Web Application Logic)

* Functionality:
  + Initialization: Imports necessary libraries, sets up the Streamlit web interface, and initializes the ResponseLLM class.
  + File Upload: Creates a file upload component in the UI, handling file selection and validation.
  + User Input: Provides a text input area for users to ask questions or provide context.
  + Submit Handling: Upon user submission:
    - Reads the uploaded file content.
    - Calls appropriate functions from the ResponseLLM class for code analysis.
    - Renders the LLM-generated review output in the UI.
* Dependencies: streamlit, src.helper (for the ResponseLLM class)

2.2 helper.py (LLM Interaction and Processing)

* Functionality:
  + Class ResponseLLM: (See Class Design section below)
* Dependencies: os, langchain\_community.document\_loaders, langchain\_core.prompts, langchain.text\_splitter, langchain\_community.vectorstores, langchain\_groq, dotenv, streamlit, langchain\_together.embeddings, langchain\_ai21

### 3. Class Design

3.1 ResponseLLM (in helper.py)

* Attributes:
  + model: Instance of the Llama 3 model (using ChatGroq).
  + embeddings: Instance of the embedding model (using AI21Embeddings).
* Methods:
  + \_\_init\_\_(self, model=None, embeddings=None): Constructor, initializes the LLM and embedding models.
  + load\_text(self, text): Loads and preprocesses text data from the uploaded code file using TextLoader.
  + split\_text(self, documents): Splits the code into smaller chunks using RecursiveCharacterTextSplitter to accommodate the LLM's context window.
  + create\_embeddings(self, split\_text): Creates embeddings for each code chunk using the AI21Embeddings model and stores them in the FAISS vector store.
  + response(self, user\_question, vector\_store):
    - Performs similarity search in the FAISS vector store based on the user's question to retrieve relevant code chunks.
    - Constructs a prompt using the user's question, retrieved context, and a predefined prompt template.
    - Sends the prompt to the Llama 3 model.
    - Returns the LLM-generated response.
  + new\_uploaded\_file(self, text, user\_question): (This function seems to duplicate the core logic in response. Consider refactoring for better organization).

### 4. Function Design

(Provide detailed descriptions of the logic within key functions, similar to the level of detail in the "Code Snippets" section from the previous Detailed Project Report. This would involve explaining the purpose, inputs, processing steps, and outputs of each significant function.)

### 5. Data Structures

* Code Input: Raw text representing the uploaded code file.
* Split Code: List of strings, where each string is a code chunk.
* Embeddings: Numerical vector representations of code chunks (likely a matrix or a list of vectors).
* FAISS Index: Structure representing the vector store, used for efficient similarity search.
* User Query: Raw text string representing the user's question or context.
* LLM Prompt: Structured text input for the Llama 3 model, typically including context, instructions, and the user's question.
* LLM Response: Raw text output generated by the Llama 3 model.