## Architecture Design Document: Code Reviewer with Llama 3 LLM

### 1. Introduction

1.1 Purpose

This Architecture Design Document (ADD) outlines the structural and behavioral design of "Code Reviewer", a web application designed to leverage the Llama 3 large language model (LLM) for automated code review and feedback. The ADD defines the system's components, their interactions, and the underlying technologies used.

1.2 Scope

This document covers the following aspects of the Code Reviewer architecture:

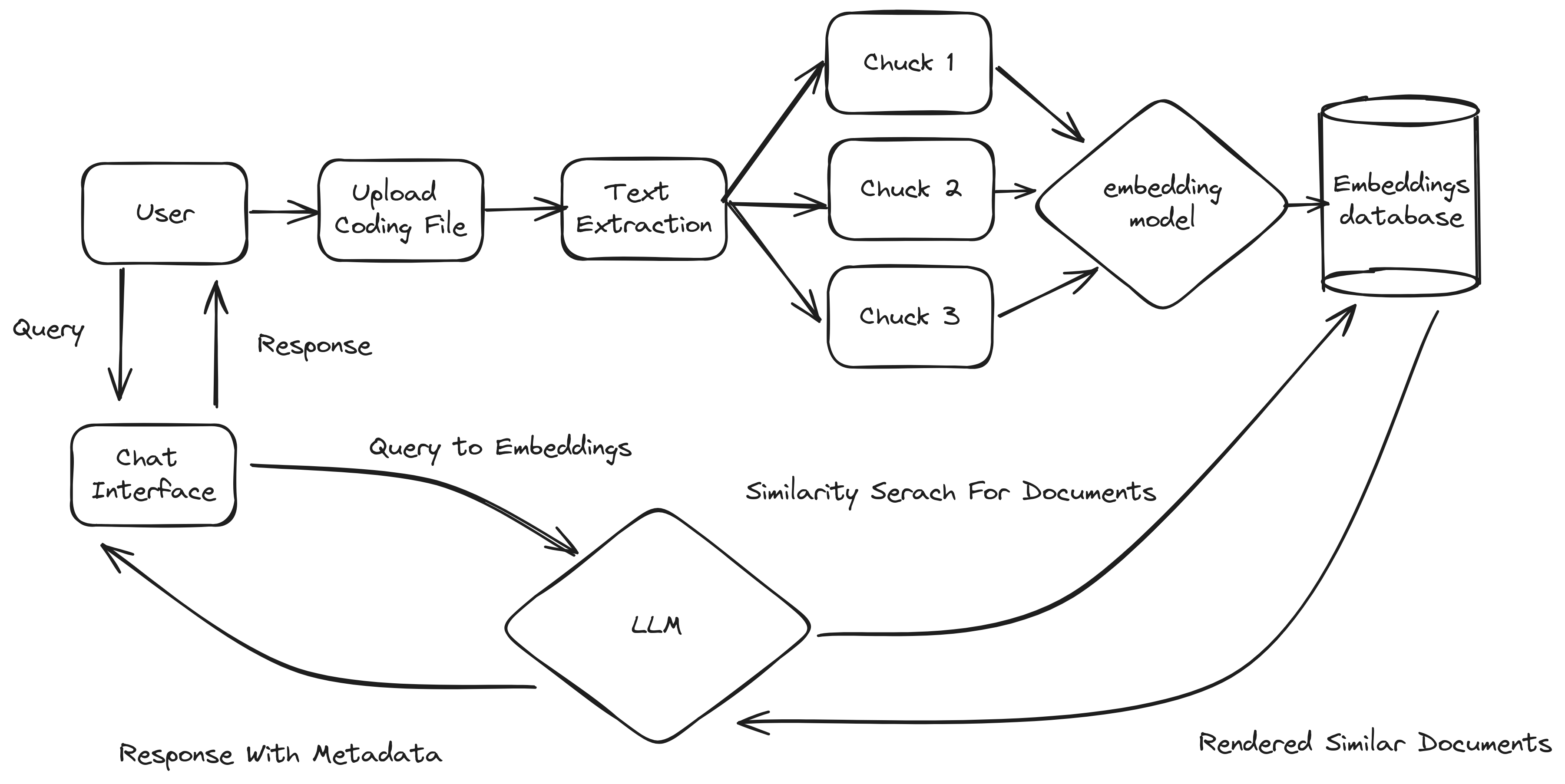
* Overall System Architecture: Describing the high-level structure and component relationships.
* Component Design: Detailing the functionality and responsibilities of each component.
* Data Flow: Illustrating how data is exchanged and processed within the system.
* Deployment Considerations: Outlining potential deployment environments and configurations.

### 2. System Architecture

2.1 Overview

Code Reviewer follows a client-server model, with a web-based client interface (Streamlit) communicating with a backend server responsible for code processing and LLM interactions. The system leverages cloud-based services and open-source libraries to provide a scalable and efficient solution.

2.2 Architecture Diagram



2.3 Components

* Client (Web UI):
  + Technology: Streamlit
  + Responsibilities:
    - Provides a user-friendly interface for code file uploads and user input (questions/context).
    - Sends user requests to the Backend (API) and displays the received review output.
* Backend (API):
  + Technology: Python (using a framework like Flask or FastAPI)
  + Responsibilities:
    - Receives code files and user input from the Client.
    - Handles file management (storage, retrieval).
    - Manages LLM interactions through the ResponseLLM class.
    - Returns processed review feedback to the Client for display.
* ResponseLLM Class (LLM Interaction):
  + Technology: Python (leveraging LangChain, AI21Embeddings, Groq, FAISS)
  + Responsibilities:
    - Loads and preprocesses code files.
    - Generates embeddings for code chunks using AI21Embeddings.
    - Manages the FAISS vector store for efficient code chunk retrieval.
    - Constructs prompts incorporating user queries and relevant code context.
    - Sends queries to the Llama 3 model via the Groq API.
    - Processes and structures the LLM's response for return to the Backend.
* Llama 3 Model (Code Analysis):
  + Technology: Llama 3 (70B parameter model) accessed through the Groq API.
  + Responsibilities:
    - Receives code and user queries embedded within prompts.
    - Analyzes the code and generates textual feedback and suggestions based on the prompt.
* AI21Embeddings:
  + Responsibilities:
    - Converts both code and natural language queries into numerical vector representations (embeddings) that can be processed by the Llama 3 model.
* FAISS Vector Store:
  + Responsibilities:
    - Stores code chunk embeddings generated by AI21Embeddings.
    - Enables efficient similarity search to retrieve contextually relevant code chunks based on user queries.

### 3. Data Flow

1. Code Upload and Input:
   * The user uploads a code file and (optionally) provides a question/context via the Streamlit UI.
2. Client-Backend Communication:
   * The Streamlit client sends the code file and user input to the Backend (API) as a request.
3. Code Preprocessing and Embedding:
   * The Backend receives the code, performs any necessary preprocessing, and passes it to the ResponseLLM class.
   * ResponseLLM splits the code into manageable chunks and uses AI21Embeddings to generate embeddings for each chunk.
4. Vector Store Management:
   * ResponseLLM adds the code chunk embeddings to the FAISS vector store.
5. Query Processing and Context Retrieval:
   * The user's query is received by the backend and passed to ResponseLLM.
   * ResponseLLM uses AI21Embeddings to generate an embedding for the query.
   * It then performs a similarity search in the FAISS vector store using the query embedding to retrieve contextually relevant code chunks.
6. Prompt Construction and LLM Query:
   * ResponseLLM constructs a prompt that includes the user's query and the retrieved code context.
   * The prompt is sent to the Llama 3 model via the Groq API.
7. Review Generation and Return:
   * Llama 3 processes the prompt and generates a text response containing the code review feedback.
   * The response is sent back to the ResponseLLM class.
8. Response Processing and Display:
   * ResponseLLM performs any necessary post-processing on the LLM's response.
   * The processed review output is returned to the Backend (API).
   * The Backend sends the review output back to the Streamlit client.
   * The Streamlit client renders the review output for the user.

### 4. Deployment Considerations

* Cloud-Based Deployment: A cloud platform (e.g., AWS, Google Cloud, Azure) is well-suited for Code Reviewer, offering:
  + Scalability: Handling varying user traffic and code analysis demands.
  + Managed Services: Simplified management of databases, storage, and API gateways.
  + Cost-Effectiveness: Pay-as-you-go models can optimize expenses.
* Containerization (e.g., Docker): Packaging the application and its dependencies into containers can enhance portability and streamline deployment.
* API Gateway: Using an API Gateway (provided by cloud platforms or dedicated solutions like Kong) can improve security, manage request routing, and enforce rate limits.

### 5. Non-Functional Requirements

* Performance: Optimize code processing, embedding generation, and LLM query time to ensure a responsive user experience.
* Scalability: Design the architecture to handle increasing numbers of users, code repositories, and analysis requests.
* Security: Implement appropriate security measures for user authentication, data storage, and communication between components.
* Maintainability: Employ modular design principles, coding standards, and documentation to facilitate code maintenance and updates.

### 6. Future Considerations

* Version Control Integration: Explore integrating Code Reviewer with popular VCS platforms (GitHub, GitLab, Bitbucket) to enable analysis within development workflows.
* Multilingual Support: Expand support to include more programming languages beyond the initial set.
* Enhanced Visualizations: Incorporate code visualizations to highlight problematic areas within the codebase.
* User Customization: Allow users to tailor review criteria, feedback preferences, and potentially integrate with their coding style guides.

This Architecture Design Document provides a high-level blueprint for the Code Reviewer application. It's essential to further refine this design during detailed design and implementation phases.