# Contents

# 1. Introduction

Large Language Models (LLMs) like ChatGPT are impressive, but they still have some major limitations. They often lack access to private or up-to-date information, and they are limited by how much text they can process at once. Retrieval-Augmented Generation (RAG) helps solve these problems by connecting the AI to a specific set of external documents.

RAG works by giving the AI the right context before it answers. When a user asks a question, the system searches through a knowledge base to find relevant information. It then combines this information with the original question, allowing the model to give an answer that is more accurate and based on facts rather than just its training data.

For this project, our group created a RAG system with a user interface built in JavaFX. This application lets users upload and organize their own files and articles. When a user asks a question, the system uses semantic search to find the best matching documents and uses them to help the LLM generate a response.

The rest of this report details how we designed and built this system.

# 2. Problem Definition & Objectives

## 2.1. Problem Definition

While Large Language Models (LLMs) are powerful tools, they face several significant limitations when used in real-world situations. First, their knowledge is limited by their training data, meaning they cannot answer questions about events that happened after they were trained. Second, they have a "context window" limit, which restricts how much information they can process at one time. Finally, general models lack specific knowledge about private organizations or specialized fields of study.

These limitations lead to four main problems:

- **Hallucination**: LLMs often confidently generate false information when they don't know the answer.
- **No Sources**: It is usually impossible to verify where the model got its information.
- **No Private Data**: Organizations cannot easily use these models to search their own internal documents.
- **Static Knowledge**: The models cannot learn new information without going through an expensive retraining process.

These issues are especially difficult in an educational setting. Students often need to find specific answers within their course materials, lecture notes, and textbooks. A standard LLM cannot access these files, so when a student asks a specific question about their class, the model often provides a generic or irrelevant answer instead of the correct one based on the syllabus.

### 2.2. Purpose & Objectives

The main goal of this project is to build a Retrieval-Augmented Generation (RAG) system that solves the problems mentioned above. By combining a standard AI model with a smart search tool, the system can provide answers that are accurate and specific to the user's documents.

**Primary Objectives:**

1. **Manage Knowledge Bases**: Allow users to easily create and organize different groups of documents (sessions) for various topics or subjects.
2. **Smart Search**: Implement a vector search that finds the right information based on the meaning of a question, rather than just matching specific keywords.
3. **Conversation Memory**: Ensure the system remembers previous messages in the chat so it can handle follow-up questions correctly.
4. **Better Ranking**: Use a re-ranking method to double-check the search results, ensuring the AI only sees the most relevant text.
5. **Show Sources**: Display exactly which document the AI used to generate its answer, allowing the user to verify the facts.
6. **Model Selection**: Give users the option to switch between different AI models (like GPT-4o or newer versions) depending on their needs.

**Secondary Objectives:**

1. **User Interface**: Build a simple, easy-to-use graphical interface using JavaFX so that non-technical users can interact with the system.
2. **Save Data**: Implement a storage system so that uploaded documents and chat history are saved and don't need to be re-processed every time the app is opened.
3. **Topic Organization**: Ensure that different sessions are kept separate so information from one subject doesn't get mixed up with another.

## 3. Target Audience

This system is designed to help anyone who needs to find specific information within a large set of documents. Most of these users are not AI experts, so the interface needs to be simple and easy to use without requiring coding skills. They value accuracy over creativity; they need to know exactly where an answer came from (citations). They are working with private or specialized documents that public AI models (like standard ChatGPT) cannot see.

- **Students**: Who need to study course materials, search through lecture notes, and find answers for assignments.
- **Researchers**: Who need to quickly find information across many technical papers and articles.
- **Professionals**: Who need to check internal company documents, manuals, or policy guides that aren't available to the public.

# 4. System Scope

The implemented RAG system encompasses the following functional scope:

1. **Session Management**:
   - Create multiple independent knowledge bases (sessions)
   - Edit session names and associated AI models
   - Delete sessions with confirmation safeguards
   - Persistent storage of session metadata
2. **Knowledge Base Management**:
   - Import documents in plain text (Text, Markdown)
   - Remove documents from knowledge bases
   - Auto index documents into vector stores using OpenAI embeddings
3. **Intelligent Query Processing**:
   - Semantic search using OpenAI embeddings (text-embedding-3-small)
   - Session-aware query contextualization
   - Hybrid re-ranking combining semantic and lexical signals
   - Retrieval of top-5 most relevant document segments
   - Source attribution for transparency
4. **Sessional Interface**:
   - Session history persistence across sessions
   - Clear session functionality
   - Real-time response generation with loading indicators
   - Display of source documents for each response
5. **Performance Optimization**:
   - Embedding caching to avoid redundant API calls
   - Incremental indexing (only process new files)
   - Session-isolated embedding stores for memory efficiency

The following features are out of scope for the current implementation:

- Web-based deployment (desktop application only)
- Multi-user collaboration features
- Document editing capabilities within the application
- Support for non-plain text document formats.
- Custom embedding model training
- Integration with databases or enterprise systems

# 5. Requirement Specifications

This section delineates the comprehensive functional and non-functional requirements that guided the development of the RAG system. The requirements were derived from user needs analysis and technical feasibility considerations.

## 5.1. Functional Requirements (FR)

1. **Session Management**
   - FR-01: The system shall allow users to create multiple independent sessions (knowledge bases) to keep different topics separate.
   - FR-02: Users must be able to rename existing sessions to better organize their work.
   - FR-03: The system shall provide a “Delete Session” function with a confirmation prompt to prevent accidental data loss.
   - FR-04: The system must automatically save the list of sessions and their settings so they remain available after closing the application.
2. **Document & Knowledge Base Management**
   - FR-05: Users must be able to import documents into a selected session.
   - FR-06: The system shall support the ingestion of plain text files (.txt) and Markdown files (.md).
   - FR-07: The system must be able to remove specific documents from a knowledge base if the user decides they are no longer needed.
   - FR-08: Upon importing, the system must automatically convert document text into vector embeddings using the OpenAI text-embedding-3-small model.
3. **Search & Query Processing**
   - FR-09: The system shall accept natural language queries from the user via a chat interface.
   - FR-10: The system must perform a semantic search to retrieve the top-5 most relevant document segments based on the user’s query.
   - FR-11: The system shall implement a re-ranking mechanism to sort retrieved results by combining semantic similarity (meaning) with lexical matching (keywords).
   - FR-12: The system must allow users to select their preferred generation model (e.g., GPT-4o, GPT-4.1) for answering questions.
4. **Response Generation & Interface**
   - FR-14: The system must clearly display the specific source documents used to generate each answer (Source Attribution).
   - FR-15: The system shall maintain a history of the current conversation, allowing users to ask follow-up questions that refer to previous answers.
   - FR-16: Users must be able to clear the current chat history to start a fresh conversation within the same session.

## 5.2. Non-Functional Requirements (NFR)

1. **Usability**

- NFR-01: The Graphical User Interface (GUI) shall be built using JavaFX to ensure a consistent desktop experience.
- NFR-02: The interface must be intuitive for non-technical users, requiring no coding knowledge to upload files or manage sessions.
- NFR-03: Visual indicators (such as progress bars) must be displayed while the system is processing embeddings or generating responses.

2. **Performance & Efficiency**

- NFR-04: The system shall implement caching for document embeddings. If a file has already been processed, the system should not re-calculate the vectors to save time and API costs.
- NFR-05: The indexing process must be incremental, meaning the system only processes new files added to a session rather than rebuilding the entire database every time.

3. **Reliability & Accuracy**

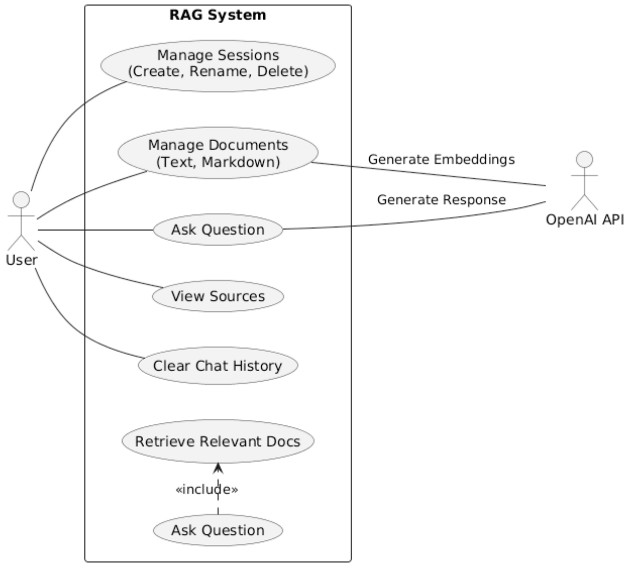
- NFR-07: The system must prioritize factual accuracy by strictly grounding responses in the provided documents to minimize hallucinations.
- NFR-08: Search results must be isolated by session; querying a session must never retrieve documents from another session.

4. **System Constraints**

- NFR-09: The application shall run as a standalone desktop application.
- NFR-10: The system requires an active internet connection to communicate with OpenAI's API for embedding and generation.

# 6. Use Case Diagram & Specifications

Shown below are the use case diagrams for the main functionalities of the RAG system, along with brief descriptions of each use case.



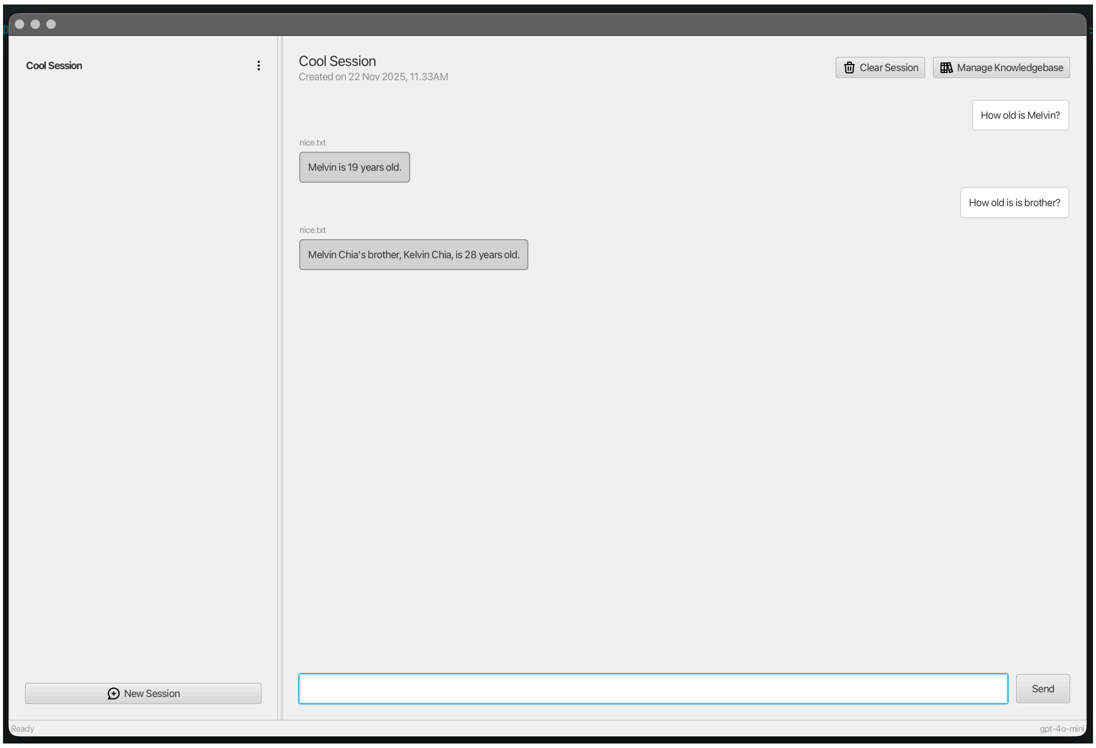
### 6.1. Actors

- **User**: The person interacting with the app (Student, Researcher, etc.).
- **OpenAI API**: The external service that provides the intelligence (embeddings and answers).

### 6.2. Main Actions (Use Cases)

- **UC01: Manage Sessions**: Covers creating, renaming, and deleting knowledge bases.
- **UC02: Upload Documents**: Adding files to the system. This connects to the API because the system needs to create "embeddings" (mathematical representations) of the text.
- **UC03: Ask Question**: The main feature. It includes the internal step of "Retrieving Relevant Docs" before sending the prompt to the API to get an answer.
- **UC04: View Sources**: Verifying where the information came from.
- **UC05: Clear Chat History**: Resetting the conversation.

## 7. UI Design



Cool Session

Created on 22 Nov 2025, 11.33AM

How old is Melvin?

nice.txt

Melvin is 19 years old.

How old is is brother?

nice.txt

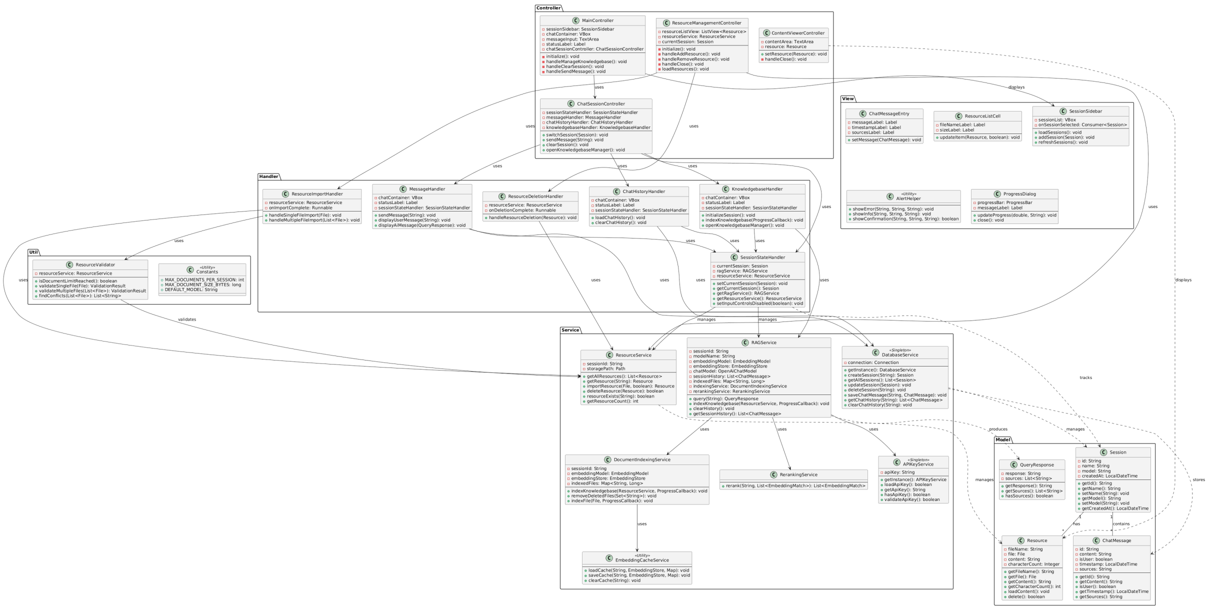
Melvin Chia's brother, Kelvin Chia, is 28 years old.

The interface is designed to be simple and user-friendly. The left sidebar allows users to manage their sessions. The main area displays the chat history, where users can see their questions and the AI's responses in a conversational format. The bottom input box is where users type their questions. The "Manage Knowledge Base" button opens a dialog for uploading and organizing documents within the selected session.

Due to time constraint and the lack of experiences of group members, we have decided not to add too much CSS styling to keep the focus on functionality and clarity. However, continuous efforts will still be made to improve the system in future iterations.

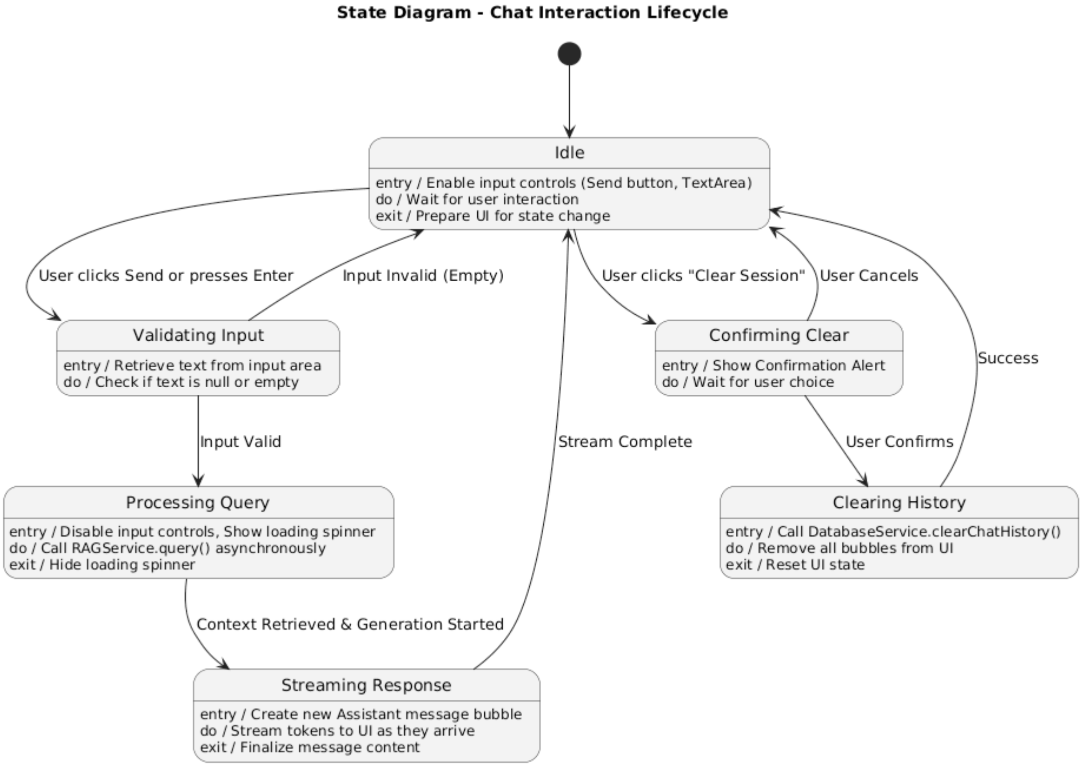
## 8. Class Diagram & OOP Design

Shown below is the class diagram extracted from the codebase, illustrating the main components of the RAG system and their relationships.

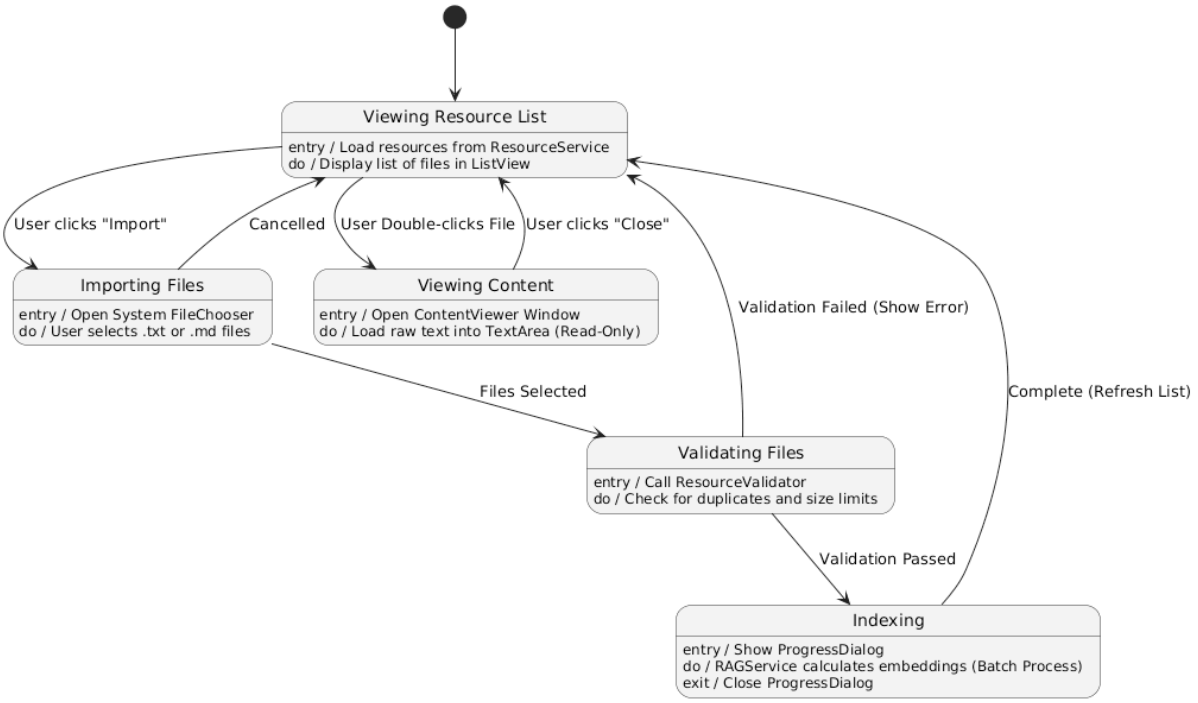

Controller
MainController
sessionSidebar: SessionSidebar
chatContainer: VBox
messageInput: TextArea
statusLabel: Label
chatSessionController: ChatSessionController
initialize(): void
handleManageKnowledgebase(): void
handleClearSession(): void
handleSendMessage(): void
ResourceManagementController
resourceListView: ListView<Resource>
resourceService: ResourceService
currentSession: Session
initialize(): void
handleAddResource(): void
handleRemoveResource(): void
handleClose(): void
loadResources(): void
ContentViewerController
contentArea: TextArea
resource: Resource
setResource(Resource): void
handleClose(): void
ChatSessionController
sessionStateHandler: SessionStateHandler
messageHandler: MessageHandler
chatHistoryHandler: ChatHistoryHandler
knowledgebaseHandler: KnowledgebaseHandler
switchSession(Session): void
sendMessage(String): void
clearSession(): void
openKnowledgebaseManager(): void
View
ChatMessageEntry
messageLabel: Label
timestampLabel: Label
sourcesLabel: Label
setMessage(ChatMessage): void
ResourceListCell
fileNameLabel: Label
sizeLabel: Label
updateItem(Resource, boolean): void
SessionSidebar
sessionList: VBox
onSessionSelected: Consumer<Session>
loadSessions(): void
addSession(Session): void
refreshSessions(): void
«Utility»
AlertHelper
showError(String, String, String): void
showInfo(String, String, String): void
showConfirmation(String, String, String): boolean
ProgressDialog
progressBar: ProgressBar
messageLabel: Label
updateProgress(double, String): void
close(): void
Handler
ResourceImportHandler
resourceService: ResourceService
onImportComplete: Runnable
handleSingleFileImport(File): void
handleMultipleFileImport(List<File>): void
MessageHandler
chatContainer: VBox
statusLabel: Label
sessionStateHandler: SessionStateHandler
sendMessage(String): void
displayUserMessage(String): void
displayAiMessage(QueryResponse): void
ResourceDeletionHandler
resourceService: ResourceService
onDeletionComplete: Runnable
handleResourceDeletion(Resource): void
ChatHistoryHandler
chatContainer: VBox
statusLabel: Label
sessionStateHandler: SessionStateHandler
loadChatHistory(): void
clearChatHistory(): void
KnowledgebaseHandler
chatContainer: VBox
statusLabel: Label
sessionStateHandler: SessionStateHandler
initializeSession(): void
indexKnowledgebase(ProgressCallback): void
openKnowledgebaseManager(): void
SessionStateHandler
currentSession: Session
ragService: RAGService
resourceService: ResourceService
setCurrentSession(Session): void
getCurrentSession(): Session
getRagService(): RAGService
getResourceService(): ResourceService
setInputControlsDisabled(boolean): void
Util
ResourceValidator
resourceService: ResourceService
isDocumentLimitReached(): boolean
validateSingleFile(File): ValidationResult
validateMultipleFiles(List<File>): ValidationResult
findConflicts(List<File>): List<String>
«Utility»
Constants
MAX_DOCUMENTS_PER_SESSION: int
MAX_DOCUMENT_SIZE_BYTES: long
DEFAULT_MODEL: String
Service
ResourceService
sessionId: String
storagePath: Path
getAllResources(): List<Resource>
getResource(String): Resource
importResource(File, boolean): Resource
deleteResource(Resource): boolean
resourceExists(String): boolean
getResourceCount(): int
RAGService
sessionId: String
modelName: String
embeddingModel: EmbeddingModel
embeddingStore: EmbeddingStore
chatModel: OpenAiChatModel
sessionHistory: List<ChatMessage>
indexedFiles: Map<String, Long>
indexingService: DocumentIndexingService
rerankingService: RerankingService
query(String): QueryResponse
indexKnowledgebase(ResourceService, ProgressCallback): void
clearHistory(): void
getSessionHistory(): List<ChatMessage>
«Singleton»
DatabaseService
connection: Connection
getInstance(): DatabaseService
createSession(String): Session
getAllSessions(): List<Session>
updateSession(Session): void
deleteSession(String): void
saveChatMessage(String, ChatMessage): void
getChatHistory(String): List<ChatMessage>
clearChatHistory(String): void
DocumentIndexingService
sessionId: String
embeddingModel: EmbeddingModel
embeddingStore: EmbeddingStore
indexedFiles: Map<String, Long>
indexKnowledgebase(ResourceService, ProgressCallback): void
removeDeletedFiles(Set<String>): void
indexFile(File, ProgressCallback): void
RerankingService
rerank(String, List<EmbeddingMatch>): List<EmbeddingMatch>
«Singleton»
APIKeyService
apiKey: String
getInstance(): APIKeyService
loadApiKey(): boolean
getApiKey(): String
hasApiKey(): boolean
validateApiKey(): boolean
«Utility»
EmbeddingCacheService
loadCache(String, EmbeddingStore, Map): void
saveCache(String, EmbeddingStore, Map): void
clearCache(String): void
Model
QueryResponse
response: String
sources: List<String>
getResponse(): String
getSources(): List<String>
hasSources(): boolean
Session
id: String
name: String
model: String
createdAt: LocalDateTime
getId(): String
getName(): String
setName(String): void
getModel(): String
setModel(String): void
getCreatedAt(): LocalDateTime
Resource
fileName: String
file: File
content: String
characterCount: Integer
getFileName(): String
getFile(): File
getContent(): String
getCharacterCount(): int
loadContent(): void
delete(): boolean
ChatMessage
id: String
content: String
isUser: boolean
timestamp: LocalDateTime
sources: String
getId(): String
getContent(): String
isUser(): boolean
getTimestamp(): LocalDateTime
getSources(): String
uses
validates
manages
displays
tracks
produces
stores
has
contains
1
*

## 9. UI Event Handling Specification

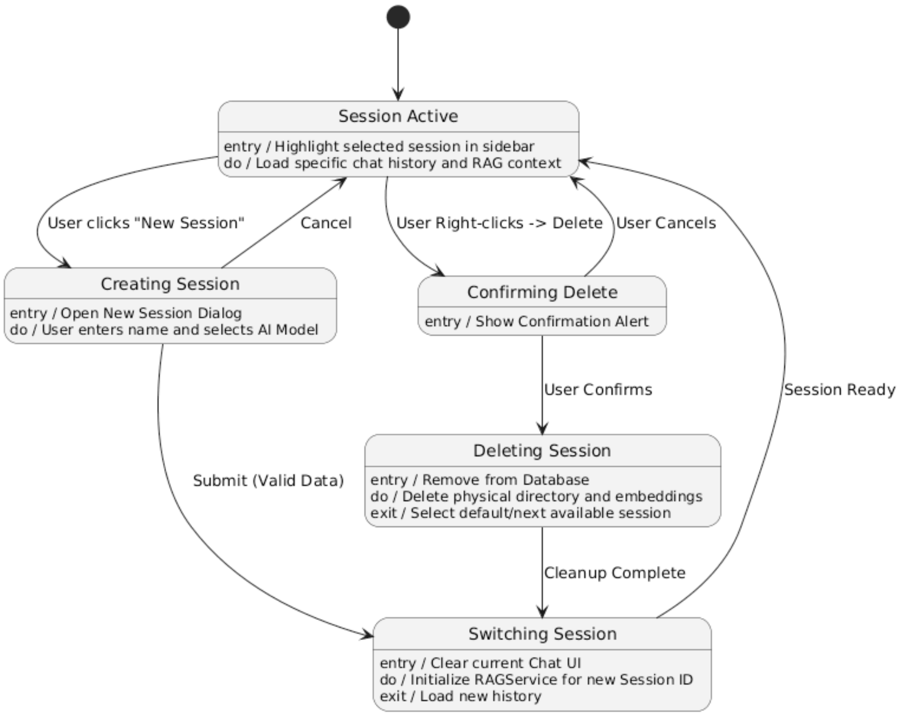
Shown below is the specification of the main UI event handling logic for the RAG system in the form of state diagrams.


State Diagram - Chat Interaction Lifecycle
Idle
entry / Enable input controls (Send button, TextArea)
do / Wait for user interaction
exit / Prepare UI for state change
User clicks Send or presses Enter
Input Invalid (Empty)
User clicks "Clear Session"
User Cancels
Validating Input
entry / Retrieve text from input area
do / Check if text is null or empty
Input Valid
Confirming Clear
entry / Show Confirmation Alert
do / Wait for user choice
Success
User Confirms
Stream Complete
Processing Query
entry / Disable input controls, Show loading spinner
do / Call RAGService.query() asynchronously
exit / Hide loading spinner
Clearing History
entry / Call DatabaseService.clearChatHistory()
do / Remove all bubbles from UI
exit / Reset UI state
Context Retrieved & Generation Started
Streaming Response
entry / Create new Assistant message bubble
do / Stream tokens to UI as they arrive
exit / Finalize message content

**State Diagram - Resource Management Flow**


Viewing Resource List
entry / Load resources from ResourceService
do / Display list of files in ListView
User clicks "Import"
Cancelled
User Double-clicks File
User clicks "Close"
Importing Files
entry / Open System FileChooser
do / User selects .txt or .md files
Viewing Content
entry / Open ContentViewer Window
do / Load raw text into TextArea (Read-Only)
Validation Failed (Show Error)
Files Selected
Complete (Refresh List)
Validating Files
entry / Call ResourceValidator
do / Check for duplicates and size limits
Validation Passed
Indexing
entry / Show ProgressDialog
do / RAGService calculates embeddings (Batch Process)
exit / Close ProgressDialog

**State Diagram - Session Management Flow**


Session Active
entry / Highlight selected session in sidebar
do / Load specific chat history and RAG context
User clicks "New Session"
Cancel
User Right-clicks -> Delete
User Cancels
Creating Session
entry / Open New Session Dialog
do / User enters name and selects AI Model
Confirming Delete
entry / Show Confirmation Alert
User Confirms
Session Ready
Deleting Session
entry / Remove from Database
do / Delete physical directory and embeddings
exit / Select default/next available session
Submit (Valid Data)
Cleanup Complete
Switching Session
entry / Clear current Chat UI
do / Initialize RAGService for new Session ID
exit / Load new history

# 10. Data Storage Strategy

## 10.1. Persistent Data Storage (Database)

For data that requires relationships and frequent updates, we used a lightweight SQL database (SQLite) managed by the DatabaseService class.

**Session Metadata:** Stores information about each knowledge base, such as the Session ID, Name, Creation Date, and the selected AI Model.

**Chat History:** Stores the conversation logs. Each message is linked to a specific Session ID, allowing the system to load the correct history when a user switches topics.

**Why SQLite** We deliberately chose SQLite over server-based alternatives because it requires zero configuration, consumes minimal system resources, and allows the entire application state to be portable (single file), which is ideal for a standalone desktop application.

## 10.2. File System Storage (File I/O)

Large text files and mathematical vectors are stored directly on the user's hard drive. The ResourceService manages this process using the Java NIO (Non-blocking I/O) package.

**Document Repository:** When a user uploads a file, it is copied directly into a dedicated directory structure: knowledgebase_storage/{sessionId}/. This ensures isolation between different sessions.

**Vector Embeddings:** The mathematical representations of the text (embeddings) are computationally expensive to create. To save time, the system serializes these vectors and saves them to a local file using the EmbeddingCacheService.

Why this approach? Storing large blocks of text and binary data in a SQL database can slow it down (bloat). Keeping them as physical files is faster for the operating system to read and easier to manage using standard file explorers.

## 10.3. In-Memory Data Structures (Runtime)

While the application is running, we utilize various ArrayList structures with dynamic sizes to manage temporary data. These collections are used throughout the runtime for tasks such as holding conversation context, storing retrieved search results, and managing UI elements, allowing the system to efficiently adapt to changing data requirements without fixed size limitations.

## 10.4. Data Validation

Before any data is stored or processed, it passes through the ResourceService logic to prevent errors.

**File Type Validation:** The system strictly accepts only Text Files (.txt) and Markdown Files (.md, .mdx). Unsupported formats are rejected immediately to prevent processing errors.

**Size Limits:** To prevent memory overflows, files are checked against a maximum size limit (defined in Constants).

**Duplicate Detection:** The system checks if a file with the same name already exists in the target directory (knowledgebase_storage/{sessionId}/). If a duplicate is found, an IOException is thrown to prevent accidental overwriting of existing knowledge.

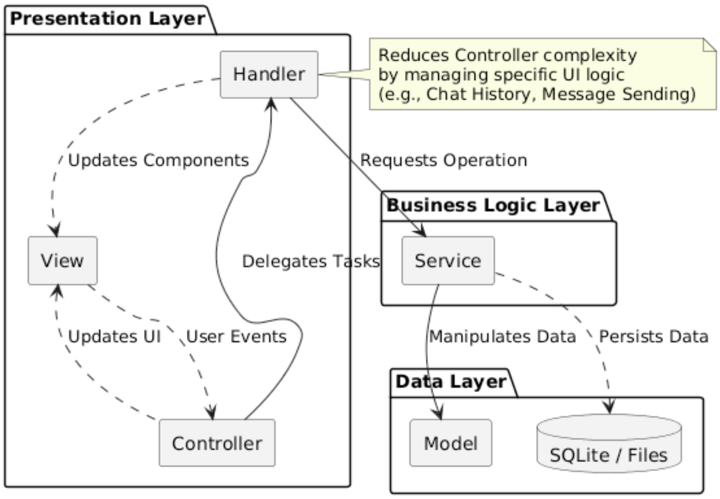
# 11. Architecture Overview

The system is architected using a standard **Model-View-Controller (MVC)** pattern, extended with a dedicated Service Layer and a Handler Pattern. This layered approach ensures a clean separation of concerns, making the application modular, testable, and easier to maintain.

## 11.1. Architectural Pattern

The application logic is divided into five distinct layers:

- **Model (Data Layer):** Represents the core business objects and data structures.
- **View (Presentation Layer):** Handles the visual representation and user interface components.
- **Controller (Orchestration Layer):** Acts as the intermediary, capturing user input and coordinating updates between the Model and View.
- **Handler (Interaction Logic):** A specialized sub-layer that manages specific UI behaviors to prevent Controllers from becoming monolithic ("bloated").
- **Service (Business Logic):** Encapsulates the complex background operations (AI processing, Database I/O, File Management).



## 11.2. Component Breakdown

1. **Model Package (`dev.assignment.model`)**

   This package contains classes that define the data structure of the application. These classes are devoid of business logic and serve primarily as data carriers.

   **Key Classes:** `Session`, `ChatMessage`, `Resource`, `QueryResponse`.

2. **View Package (`dev.assignment.view`)**

   This package encompasses all custom UI components and visual helpers built with JavaFX. It is responsible for rendering the interface but contains no business logic.

   **Key Components:**

   - `SessionSidebar`: Manages the visual list of available knowledge bases.
   - `ChatMessageEntry`: A custom cell used to render user and AI messages.
   - `AlertHelper`: A utility for displaying standard system dialogs.

3. **Controller Package (`dev.assignment.controller`)**

   The Controllers act as the entry points for the FXML views. They initialize the view state and bind UI elements to the backend logic.

   **Key Classes:**

   - `ChatSessionController`: The main controller for the chat interface.
   - `ResourceManagementController`: Controls the document upload and management window.

4. **Handler Package (`dev.assignment.handler`)**

   To strictly adhere to the Single Responsibility Principle (SRP), complex logic was extracted from the main Controllers into specialized Handlers. This prevents the "God Class" anti-pattern where a single Controller handles unrelated tasks.

   **Key Handlers:**

   - `MessageHandler`: strictly handles sending messages and updating the chat bubble UI.
   - `ChatHistoryHandler`: Manages loading and clearing previous conversations.
   - `SessionStateHandler`: Manages the active state (e.g., locking buttons while AI is thinking).

5. **Service Package (`dev.assignment.service`)**

   This is the "engine room" of the application. It contains all the heavy computational logic, database operations, and external API calls. Controllers and Handlers call these services to perform tasks.

   **Key Services:**

   - `RAGService`: The core coordinator that combines retrieval and generation.
   - `DocumentIndexingService`: Handles reading files and converting them into vector embeddings.
   - `DatabaseService`: Manages the SQLite connection for persistent storage.

## 12. Development Tools & Technologies

We utilized the following software and libraries to design, implement, and document the RAG system:

### 12.1. Core Frameworks & Libraries

- **JavaFX:** The primary framework used for building the Graphical User Interface (GUI).
- **LangChain4j:** The library that provided the foundational RAG functionalities, including vector stores and LLM integrations.

- **Log4j:** For logging application events and errors.
- **Visual Studio Code (VSCode):** Our primary Text Editor and Integrated Development Environment (IDE) for writing and managing the Java codebase.
- **SceneBuilder:** For visually designing JavaFX FXML layouts.

### 12.2. Documentation

- **Typst:** For creating this technical report with structured formatting.
- **PlantUML:** For generating class diagrams and state diagrams from textual descriptions.

## 13. Project Timeline

1. **Requirement Specification (Week 5)**
   - Drafted requirement specifications and use case diagrams.
   - Defined system scope and created architecture diagrams.
2. **UI Design & Prototyping (Week 6)**
   - Designed UI wireframes using SceneBuilder.
   - Created FXML layouts for main screens.
3. **Core Implementation (Weeks 7 - 9)**
   - Developed the Model, View, Controller, Handler, and Service layers.
   - Integrated LangChain4j for RAG functionalities.
   - Implemented session management, document ingestion, semantic search, and response generation.
4. **Documentation (Week 10)**
   - Created technical diagrams with PlantUML based on the codebase.
   - Compiled the technical report using Typst.