# ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

# Functional MVP for a Local LLM System: Chatbot, Code Assistant, and Research Tool

## 1. System Architecture Overview

The proposed system is a containerized, multi-functional AI assistant designed to operate entirely on local hardware, addressing privacy concerns and leveraging the capabilities of open-source Large Language Models (LLMs). The architecture is built on a microservices model, ensuring modularity and scalability, with Docker Compose orchestrating the various components. This design supports three primary use cases: a general-purpose chatbot, a specialized code assistant, and a research tool powered by a Retrieval-Augmented Generation (RAG) pipeline. The system is optimized for limited hardware, utilizing lightweight models like TinyLlama and, if resources permit, more powerful models like GPT-OSS:20b, all managed through the Ollama framework. The core principles guiding this architecture are modularity, containerization, and a local-first, privacy-centric approach, ensuring a robust, maintainable, and secure system that can be deployed and run on a personal computer or a small server.

### 1.1 Core Principles for a Lightweight MVP

The development of this Minimum Viable Product (MVP) is guided by a set of core principles designed to ensure the system is lightweight, maintainable, and suitable for deployment on resource-constrained hardware. These principles are not merely theoretical but are reflected in the choice of technologies and the architectural design. The emphasis is on creating a system that is easy to set up, manage, and extend, while prioritizing user privacy and data security. The following subsections detail the key principles that form the foundation of this MVP.

#### 1.1.1 Modularity and Microservices

The architecture is fundamentally based on the principles of modularity and microservices, a design choice that offers significant advantages in terms of flexibility, scalability, and maintainability. Instead of building a monolithic application, the system is decomposed into a collection of smaller, independent services, each responsible for a specific function. This approach is well-documented in various successful AI application deployments . For instance, a typical architecture might include a dedicated service for the language model API, another for the database, a third for the main application logic, and a fourth for the user interface . This separation of concerns allows for independent development, deployment, and scaling of each service. For example, if the research tool requires more processing power for its RAG pipeline, its service can be scaled independently without affecting the chatbot or code assistant services. This modularity also simplifies the process of updating or replacing individual components. If a more efficient vector database becomes available, the research tool's service can be updated to use it without requiring changes to the rest of the system. This design is particularly beneficial for an MVP, as it allows for incremental development and testing of each component, reducing the complexity of the initial build and facilitating future enhancements.

#### 1.1.2 Containerization with Docker and Docker Compose

Containerization is a cornerstone of the system's deployment strategy, with Docker and Docker Compose being the primary tools for packaging, distributing, and running the application. This approach ensures that the system is portable, reproducible, and easy to manage across different environments. Each microservice is encapsulated within its own Docker container, which includes all the necessary dependencies, libraries, and configuration files. This eliminates the "it works on my machine" problem, as the containerized application will run consistently regardless of the underlying host operating system. Docker Compose is used to define and orchestrate the multi-container application, allowing for the entire system to be started, stopped, and managed with a single command. This is particularly advantageous for a local deployment, as it simplifies the setup process significantly. A `docker-compose.yml` file can specify all the services, their dependencies, network configurations, and volume mounts, providing a clear and concise blueprint for the entire system . For example, the compose file can define services for the Ollama LLM engine, the vector database, the backend API, and the frontend UI, and specify how they communicate with each other over a private Docker network. This containerized approach not only simplifies deployment but also enhances security by isolating the services from each other and the host system.

#### 1.1.3 Local-First and Privacy-Centric Design

A key design principle of this MVP is its local-first and privacy-centric nature. The entire system is designed to run on local hardware, without relying on external cloud services or APIs for its core functionality. This is a significant departure from many modern AI applications and offers several important benefits. Firstly, it ensures complete data privacy and security. All user data, including chat histories, code, and research documents, remains on the user's machine and is not sent to any third-party servers. This is a critical feature for users who handle sensitive or confidential information. Secondly, it provides greater control and customization. Users can choose which LLMs to run, modify the system's behavior, and extend its capabilities without being constrained by the limitations of a cloud-based service. This is particularly relevant for developers and researchers who want to experiment with different models and techniques. The use of local LLMs, managed through Ollama, is central to this design. Models like TinyLlama and GPT-OSS:20b can be run entirely offline, providing powerful AI capabilities without the need for an internet connection . This local-first approach also has the benefit of reducing latency, as there is no network overhead for API calls to remote servers. The system is designed to be self-contained, with all necessary components, including the LLM, vector database, and application logic, running within the local Docker environment.

### 1.2 High-Level Component Interaction

The system's architecture is designed around a clear and logical flow of data and control between its various components. The user interacts with a single, unified interface, which in turn communicates with a set of specialized backend services. These services are responsible for handling the different use cases—chat, code assistance, and research—and they all rely on the Ollama framework as the central engine for running the local LLMs. This section provides a high-level overview of how these components interact to deliver the system's functionality.

#### 1.2.1 User Interface (UI) as the Central Interaction Point

The User Interface (UI) serves as the primary point of interaction for the user, providing a unified and intuitive way to access all the system's features. Instead of having separate applications for the chatbot, code assistant, and research tool, the system will feature a single, integrated web-based UI. This approach simplifies the user experience and provides a consistent look and feel across all use cases. The UI will be built using modern web technologies like React or Vue.js, and will communicate with the backend services through a RESTful API. The UI will be designed to be responsive, ensuring a seamless experience on both desktop and mobile devices. It will feature a chat-like interface, where users can type their queries and receive responses in real-time. The UI will also include specific input fields and controls for the different use cases. For example, for the code assistant, there might be a dedicated code editor and a button to trigger code generation. For the research tool, there will be an interface for uploading documents and asking questions about them. The UI will be responsible for capturing user input, sending it to the appropriate backend service, and displaying the responses in a clear and organized manner. This centralized UI design is a key aspect of the system's usability, making it easy for users to switch between different tasks without having to learn multiple interfaces.

#### 1.2.2 API Gateway for Routing Requests

To manage the communication between the UI and the various backend services, the system will employ an API Gateway. This component acts as a single entry point for all client requests, routing them to the appropriate backend service based on the nature of the request. For example, a request related to code generation would be routed to the Code Assistant Service, while a request to query a document would be routed to the Research Tool Service. This routing logic can be based on the URL path, request headers, or other parameters. The API Gateway provides several benefits. It simplifies the client-side code, as the UI only needs to know the address of the API Gateway, rather than the addresses of all the individual services. It also provides a central point for implementing cross-cutting concerns, such as authentication, authorization, logging, and rate limiting. For example, the API Gateway can be configured to require an API key for all requests, ensuring that only authorized users can access the system. It can also log all incoming requests and outgoing responses, providing valuable insights into the system's usage and performance. In a Docker-based deployment, the API Gateway can be implemented using a reverse proxy like Nginx, which is well-suited for this role and can be easily configured to route traffic to the different services . This centralized routing and management of requests is a key aspect of the system's architecture, ensuring a clean and efficient communication flow.

#### 1.2.3 Specialized Backend Services for Each Use Case

The core logic of the system is divided into a set of specialized backend services, each responsible for handling a specific use case. This microservices-based approach allows for a clean separation of concerns and makes the system more modular and maintainable. The three main backend services are the Chatbot Service, the Code Assistant Service, and the Research Tool Service. Each of these services will be implemented as a separate Docker container, and will expose a RESTful API for communication.

\* \*\*Chatbot Service\*\*: This service will handle all general-purpose conversational queries. It will receive user messages from the API Gateway, format them appropriately, and send them to the Ollama LLM for processing. It will then stream the LLM's response back to the UI in real-time. This service will also be responsible for managing the conversation history, ensuring that the LLM has the necessary context to provide coherent and relevant responses.

\* \*\*Code Assistant Service\*\*: This service will be specialized for code-related tasks. It will be able to generate code snippets, explain existing code, and help with debugging. It may use a different, code-specific LLM, such as CodeLlama, which is fine-tuned for programming tasks. The service will receive code-related queries from the UI, process them, and return the generated code or explanations. It may also integrate with other tools, such as a code execution environment, to provide more advanced functionality.

\* \*\*Research Tool Service\*\*: This service will implement the RAG pipeline, allowing users to interact with their own documents. It will handle the ingestion of documents, the generation of vector embeddings, and the storage of these embeddings in a vector database. When a user asks a question, the service will retrieve the most relevant document chunks from the vector database and use them to augment the prompt sent to the LLM, resulting in more accurate and context-aware responses.

This division of labor among specialized services is a key strength of the architecture, as it allows each service to be optimized for its specific task and scaled independently.

#### 1.2.4 Ollama as the Central LLM Engine

At the heart of the system is Ollama, which serves as the central engine for running the local LLMs. Ollama is a powerful and user-friendly framework that simplifies the process of downloading, managing, and running a wide variety of open-source LLMs. It provides a simple command-line interface and a RESTful API for interacting with the models. The backend services will communicate with the Ollama service to generate text, and Ollama will handle all the low-level details of model inference. This includes loading the model into memory, tokenizing the input, running the inference process, and streaming the output back to the client. The use of Ollama provides several advantages. It abstracts away the complexity of working with different LLM formats and inference engines, providing a unified interface for all models. It also handles the process of downloading and managing the model files, which can be quite large. The backend services can simply specify the name of the model they want to use (e.g., `tinyllama` or `gpt-oss:20b`), and Ollama will take care of the rest. This makes it easy to switch between different models or to use different models for different tasks. For example, the Code Assistant Service could use a code-specific model, while the Chatbot Service uses a general-purpose model. The Ollama service will be run as a separate Docker container, and will be accessible to the other services via the internal Docker network. This centralized management of LLMs is a key aspect of the system's design, providing a flexible and powerful foundation for all its AI capabilities.

## 2. Containerized Deployment Strategy

The deployment of this multi-functional AI assistant is designed to be as simple and reproducible as possible, leveraging the power of Docker and Docker Compose. This containerized approach ensures that the entire system, with all its dependencies, can be packaged and run on any machine that supports Docker, without the need for complex manual configuration. The strategy involves defining each component of the system as a separate service in a `docker-compose.yml` file, which orchestrates the creation, networking, and management of the containers. This section details the key aspects of this deployment strategy, including the orchestration process, service definitions, and data persistence.

### 2.1 Docker Compose Orchestration

Docker Compose is the primary tool for orchestrating the deployment of the system's various microservices. It allows for the definition of a multi-container application in a single YAML file, which can then be started, stopped, and managed with simple commands. This approach is ideal for a local deployment, as it simplifies the process of setting up and running the entire system. The `docker-compose.yml` file will define all the services that make up the application, including the web UI, the API gateway, the various backend services, the Ollama LLM engine, and the vector database. It will also specify the dependencies between these services, ensuring that they are started in the correct order. For example, the backend services will depend on the Ollama service, and the research tool service will depend on the vector database. Docker Compose will also manage the creation of a private network for the containers, allowing them to communicate with each other securely and efficiently. This orchestration approach provides a clear and concise blueprint for the entire system, making it easy to understand, manage, and replicate.

#### 2.1.1 Defining Services for Each Component

The `docker-compose.yml` file is the heart of the Docker Compose orchestration, where each component of the system is defined as a separate service. This includes the web UI, the API gateway, the specialized backend services (chatbot, code assistant, research tool), the Ollama engine, and the vector database. For each service, we can specify a variety of configuration options, such as the Docker image to use, the build context, the ports to expose, the environment variables to set, and the volumes to mount. This allows us to tailor the configuration of each service to its specific needs, ensuring that it has the necessary resources and dependencies to function correctly. By defining each component as a separate service, we can also take advantage of Docker Compose's dependency management features, ensuring that services are started in the correct order and that they can communicate with each other as needed.

#### 2.1.2 Managing Inter-Container Communication

A key aspect of the Docker Compose orchestration is the management of inter-container communication. By default, Docker Compose creates a private network for the application, and all the services are connected to this network. This allows the containers to communicate with each other using their service names as hostnames, without the need to expose their ports to the host machine. For example, the backend API service can connect to the Ollama service using the hostname `ollama` and the port `11434`, without needing to know the IP address of the Ollama container. This simplifies the configuration of the services and enhances the security of the system, as the internal communication is isolated from the outside world. The API Gateway, which will be implemented using a reverse proxy like Nginx, will be the only service that exposes its ports to the host machine. It will be configured to route incoming requests to the appropriate backend service based on the URL path. This centralized management of communication ensures a clean and efficient flow of data between the services, and makes the system more robust and secure.

#### 2.1.3 Volume Mounting for Data Persistence

To ensure that data is not lost when the containers are stopped or restarted, the deployment strategy will make extensive use of Docker volumes. Volumes are a mechanism for persisting data generated by and used by Docker containers. The `docker-compose.yml` file will define named volumes for each service that needs to persist data. For example, a volume will be created for the Ollama service to store the downloaded LLM models, so that they don't need to be downloaded every time the container is started. Another volume will be created for the vector database to store the document embeddings and metadata. A volume will also be created for the backend services to store user data, such as chat histories and session information. By mounting these volumes to the appropriate directories within the containers, the data will be persisted on the host machine, even if the containers are removed. This ensures that the system can be stopped and started without losing any important data, and makes it easier to back up and restore the system's state. The use of volumes is a critical aspect of the deployment strategy, ensuring the reliability and durability of the system.

### 2.2 Service Definitions

The `docker-compose.yml` file will provide a detailed definition for each of the services that make up the system. These definitions will specify the Docker image to use, the container name, the ports to expose, the environment variables to set, and the volumes to mount. This section provides an overview of the service definitions for the key components of the system.

#### 2.2.1 Web UI Service (React/Next.js)

The Web UI service will be responsible for providing the user-facing interface of the application. It will be built using a modern JavaScript framework like React or Next.js, and will be served by a lightweight web server like Nginx. The service will be defined in the `docker-compose.yml` file to use a custom-built Docker image, which will be based on a Node.js image for the build process and an Nginx image for serving the static files. The service will expose port 80 (or another port of the user's choice) to the host machine, allowing users to access the application through their web browser. The service will also be configured to communicate with the API Gateway service to send user requests and receive responses. For development purposes, the source code directory of the UI application can be mounted as a volume in the container, allowing for hot-reloading and faster development cycles.

#### 2.2.2 API Gateway Service (FastAPI/Node.js)

The API Gateway service will act as the single entry point for all client requests, routing them to the appropriate backend service. It will be implemented using a high-performance web framework like FastAPI (for Python) or Express (for Node.js), and will be run in a separate Docker container. The service will be defined in the `docker-compose.yml` file to use a custom-built Docker image, and will be configured to listen on a specific port (e.g., 8000). The service will be responsible for implementing the routing logic, which will be based on the URL path of the incoming requests. For example, requests to `/api/chat` will be routed to the Chatbot Service, while requests to `/api/code` will be routed to the Code Assistant Service. The API Gateway will also be responsible for implementing cross-cutting concerns, such as authentication, logging, and rate limiting.

#### 2.2.3 Chatbot Service

The Chatbot Service will be a specialized backend service responsible for handling all general-purpose conversational queries. It will be implemented using a Python framework like FastAPI, and will be run in a separate Docker container. The service will be defined in the `docker-compose.yml` file to use a custom-built Docker image, and will be configured to communicate with the Ollama service to generate responses. The service will receive user messages from the API Gateway, format them into a prompt for the LLM, and send them to the Ollama API. It will then stream the LLM's response back to the API Gateway, which will in turn stream it to the UI. The service will also be responsible for managing the conversation history, which will be stored in a database or a persistent cache.

#### 2.2.4 Code Assistant Service

The Code Assistant Service will be another specialized backend service, focused on providing code-related assistance. It will be implemented using a Python framework like FastAPI, and will be run in a separate Docker container. The service will be defined in the `docker-compose.yml` file to use a custom-built Docker image, and will be configured to communicate with the Ollama service, potentially using a different, code-specific LLM. The service will receive code-related queries from the API Gateway, process them, and return the generated code or explanations. It may also integrate with other tools, such as a code execution environment or a static analysis tool, to provide more advanced functionality.

#### 2.2.5 Research Tool Service

The Research Tool Service will be the backend service responsible for implementing the RAG pipeline. It will be implemented using a Python framework like FastAPI, and will be run in a separate Docker container. The service will be defined in the `docker-compose.yml` file to use a custom-built Docker image, and will be configured to communicate with both the Ollama service and the vector database. The service will handle the ingestion of documents, the generation of vector embeddings, and the storage of these embeddings in the vector database. When a user asks a question, the service will retrieve the most relevant document chunks from the vector database and use them to augment the prompt sent to the LLM.

#### 2.2.6 Ollama Service

The Ollama service will be the central engine for running the local LLMs. It will be run in a separate Docker container, using the official Ollama Docker image. The service will be defined in the `docker-compose.yml` file to expose port 11434, which is the default port for the Ollama API. The service will also be configured to mount a volume to persist the downloaded model files. The backend services will communicate with the Ollama service via its RESTful API, sending prompts and receiving generated text. The Ollama service will handle all the low-level details of model inference, including loading the model into memory, tokenizing the input, and running the inference process.

#### 2.2.7 Vector Database Service (Faiss/ChromaDB)

The Vector Database service will be responsible for storing and retrieving the vector embeddings used by the Research Tool. It will be run in a separate Docker container, using a lightweight and efficient vector database like Faiss or ChromaDB. The service will be defined in the `docker-compose.yml` file to use the appropriate Docker image for the chosen database, and will be configured to expose a port for communication. The service will also be configured to mount a volume to persist the vector data. The Research Tool service will communicate with the Vector Database service to store the embeddings of the ingested documents and to perform similarity searches to retrieve relevant chunks for a given query.

## 3. Core Component: The Chatbot

The chatbot is a central feature of the MVP, providing a general-purpose conversational interface for users. It is designed to be a simple yet effective tool for natural language interaction, leveraging the power of local LLMs to generate coherent and context-aware responses. The architecture of the chatbot is built around a clear separation of concerns, with a dedicated user interface, a backend API for processing requests, and a direct integration with the Ollama engine for model inference. This modular design ensures that the chatbot is easy to develop, maintain, and extend.

### 3.1 Architecture and Implementation

The implementation of the chatbot is divided into three main parts: the user interface, the backend API, and the integration with Ollama. The user interface provides the conversational interface for the user, the backend API handles the logic for processing chat requests, and the integration with Ollama is responsible for generating the responses. This separation of concerns allows for a clean and organized codebase, making it easier to develop and maintain the chatbot.

#### 3.1.1 User Interface for Conversational Interaction

The user interface for the chatbot is designed to be simple and intuitive, providing a familiar chat-like experience for the user. It will be built using a modern JavaScript framework like React or Vue.js, and will feature a message history, an input field for typing messages, and a send button. The UI will be responsible for displaying the conversation history, capturing user input, and sending it to the backend API. It will also be responsible for displaying the responses from the backend API in a clear and organized manner. The UI will be designed to be responsive, ensuring a seamless experience on both desktop and mobile devices.

#### 3.1.2 Backend API for Handling Chat Requests

The backend API for the chatbot is responsible for handling the logic for processing chat requests. It will be built using a lightweight web framework like FastAPI or Node.js, and will expose a RESTful API for communication with the user interface. The API will receive user messages from the UI, format them into a prompt for the LLM, and send them to the Ollama service for processing. It will then receive the response from the Ollama service and return it to the UI. The API will also be responsible for maintaining the conversation history, which is essential for providing context-aware responses.

#### 3.1.3 Integration with Ollama's Chat API

The integration with Ollama's chat API is the core of the chatbot's functionality. The backend API will use the official Ollama Python library to communicate with the Ollama service. The library provides a simple and intuitive interface for interacting with the Ollama API, allowing the backend API to send prompts to the LLM and receive generated responses. The library also supports streaming responses, which is crucial for creating a responsive user interface where the model's output appears in real-time, token by token.

### 3.2 Leveraging Local LLMs

The chatbot leverages the power of local LLMs to generate coherent and context-aware responses. The use of local LLMs ensures that all user data remains on the user's machine, providing a high level of privacy and security. The chatbot is designed to be flexible, allowing for the use of different models depending on the user's needs and hardware capabilities.

#### 3.2.1 Using General-Purpose Models (e.g., TinyLlama)

For general-purpose conversational tasks, the chatbot will use a lightweight and efficient model like TinyLlama. TinyLlama is a small but powerful model that is well-suited for running on limited hardware. It is capable of generating coherent and context-aware responses, making it an ideal choice for the chatbot. The use of a general-purpose model like TinyLlama ensures that the chatbot can handle a wide range of conversational topics, from casual chit-chat to more complex queries.

#### 3.2.2 Model Selection and Management via Ollama

The chatbot's architecture is designed to be flexible, allowing for the easy selection and management of different models. This is achieved through the use of Ollama, which provides a simple and intuitive interface for managing local LLMs. The backend API can be configured to use different models by simply changing the model name in the API call. This allows the user to choose the model that best suits their needs, whether it's a lightweight model like TinyLlama for general-purpose chat or a more powerful model like GPT-OSS:20b for more complex tasks.

#### 3.2.3 Maintaining Conversation History and Context

To provide coherent and context-aware responses, the chatbot maintains a history of the conversation. This is achieved by storing the conversation history in a lightweight database like SQLite or by using a simple in-memory data structure. The conversation history is then passed to the LLM as part of the prompt, allowing the model to understand the context of the conversation and generate a relevant response. This is a crucial feature for creating a natural and engaging conversational experience.

## 4. Core Component: The Code Assistant

The code assistant is a specialized tool designed to help users with their coding tasks. It leverages the power of code-specific LLMs to provide accurate and relevant assistance, including code generation, explanation, and debugging. The architecture of the code assistant is similar to that of the chatbot, with a dedicated user interface, a backend API, and an integration with the Ollama engine. However, it is tailored to the specific needs of developers, with a focus on code-related queries and tasks.

### 4.1 Architecture and Implementation

The implementation of the code assistant is divided into three main parts: the user interface, the backend API, and the integration with code-specific LLMs. The user interface provides a dedicated area for code-related queries, the backend API handles the logic for processing code requests, and the integration with code-specific LLMs is responsible for generating the responses.

#### 4.1.1 User Interface for Code-Related Queries

The user interface for the code assistant is designed to be simple and intuitive, providing a dedicated area for code-related queries. It will feature a code editor for writing and editing code, an input field for typing queries, and a button to trigger code generation or explanation. The UI will be responsible for displaying the generated code or explanations in a clear and organized manner, with syntax highlighting to improve readability.

#### 4.1.2 Backend API for Processing Code Requests

The backend API for the code assistant is responsible for handling the logic for processing code requests. It will be built using a lightweight web framework like FastAPI or Node.js, and will expose a RESTful API for communication with the user interface. The API will receive code-related queries from the UI, format them into a prompt for the LLM, and send them to the Ollama service for processing. It will then receive the response from the Ollama service and return it to the UI.

#### 4.1.3 Integration with Code-Specialized LLMs

The integration with code-specialized LLMs is the core of the code assistant's functionality. The backend API will use the official Ollama Python library to communicate with the Ollama service. The library provides a simple and intuitive interface for interacting with the Ollama API, allowing the backend API to send prompts to the LLM and receive generated responses. The API will be configured to use a code-specific LLM, such as CodeLlama, to provide more accurate and relevant responses.

### 4.2 Leveraging Code-Specific LLMs

The code assistant leverages the power of code-specific LLMs to provide accurate and relevant assistance. These models are fine-tuned on a large corpus of code, making them well-suited for code-related tasks. The use of code-specific LLMs ensures that the code assistant can generate high-quality code and provide insightful explanations.

#### 4.2.1 Using Models like CodeLlama or Mistral-Instruct

For code-related tasks, the code assistant will use a code-specific LLM like CodeLlama or Mistral-Instruct. These models are fine-tuned on a large corpus of code, making them well-suited for tasks like code generation, explanation, and debugging. The use of a code-specific model ensures that the code assistant can provide accurate and relevant assistance, even for complex coding tasks.

#### 4.2.2 Generating Code Snippets and Explanations

The code assistant is capable of generating code snippets in a variety of programming languages. The user can simply type a description of the code they want to generate, and the code assistant will generate the corresponding code snippet. The code assistant can also explain existing code, providing a detailed breakdown of what the code does and how it works. This is a valuable feature for developers who are trying to understand a new codebase or debug a complex piece of code.

#### 4.2.3 Automating Boilerplate Code Generation

The code assistant can also be used to automate the generation of boilerplate code. This is a common task for developers, and it can be time-consuming and tedious. The code assistant can generate boilerplate code for a variety of tasks, such as creating a new class, setting up a database connection, or implementing a RESTful API. This can save developers a significant amount of time and effort, allowing them to focus on more complex and creative tasks.

## 5. Core Component: The Research Tool

The research tool is a powerful feature of the MVP, allowing users to interact with a collection of documents and extract information from them. It uses a Retrieval-Augmented Generation (RAG) pipeline to retrieve relevant information from the documents and then uses an LLM to generate a summary or answer. The architecture of the research tool is more complex than that of the chatbot and code assistant, as it involves a vector database for storing and retrieving document embeddings.

### 5.1 Architecture and Implementation

The implementation of the research tool is divided into three main parts: the user interface, the backend API, and the RAG pipeline. The user interface provides a way for users to upload documents and ask questions about them, the backend API handles the logic for managing research queries, and the RAG pipeline is responsible for retrieving relevant information and generating responses.

#### 5.1.1 User Interface for Document Interaction

The user interface for the research tool is designed to be simple and intuitive, providing a way for users to upload documents and ask questions about them. It will feature a file upload area for uploading documents, an input field for typing questions, and a button to trigger the search. The UI will be responsible for displaying the retrieved information and the generated summary or answer in a clear and organized manner.

#### 5.1.2 Backend API for Managing Research Queries

The backend API for the research tool is responsible for handling the logic for managing research queries. It will be built using a lightweight web framework like FastAPI or Node.js, and will expose a RESTful API for communication with the user interface. The API will receive research queries from the UI, process them using the RAG pipeline, and return the generated summary or answer to the UI.

#### 5.1.3 Retrieval-Augmented Generation (RAG) Pipeline

The RAG pipeline is the core of the research tool's functionality. It is responsible for retrieving relevant information from the documents and then using an LLM to generate a summary or answer. The pipeline consists of several steps, including document ingestion, vector embedding, vector storage and retrieval, and integration with an LLM for response generation.

### 5.2 Building the RAG Pipeline

The RAG pipeline is a complex system that involves several components working together to provide accurate and relevant responses. The following sections provide a detailed overview of the steps involved in building the RAG pipeline.

#### 5.2.1 Document Ingestion and Indexing

The first step in the RAG pipeline is to ingest the documents and index them for retrieval. This involves loading the documents from a file or a directory, splitting them into smaller chunks, and then creating a vector representation of each chunk. The vector representation is a numerical representation of the text that captures its semantic meaning.

#### 5.2.2 Vector Embeddings with Sentence Transformers

To create the vector representations of the text, the RAG pipeline uses a sentence transformer model. A sentence transformer is a type of neural network that is trained to map sentences to a high-dimensional vector space, where semantically similar sentences are close to each other. The RAG pipeline will use a pre-trained sentence transformer model to generate the vector embeddings of the document chunks.

#### 5.2.3 Vector Storage and Retrieval with Faiss

Once the vector embeddings have been generated, they need to be stored in a vector database for efficient retrieval. The RAG pipeline will use a lightweight and efficient vector database like Faiss or ChromaDB to store the vector embeddings. When a user asks a question, the RAG pipeline will generate a vector embedding of the question and then use the vector database to retrieve the most similar document chunks.

#### 5.2.4 Integration with LangChain for Orchestration

To orchestrate the various components of the RAG pipeline, the research tool will use a framework like LangChain. LangChain is a powerful framework for building applications with LLMs, and it provides a set of tools and abstractions for building RAG pipelines. LangChain will be used to manage the document ingestion, vector embedding, vector storage and retrieval, and integration with the LLM for response generation.

## 6. Local LLM Integration with Ollama

The successful implementation of a local, multi-functional AI system hinges on a robust and efficient method for managing and interacting with Large Language Models (LLMs). For this MVP, Ollama has been selected as the core engine for local LLM deployment and management. Ollama is an open-source tool designed to simplify the process of running open-source LLMs on local hardware, abstracting away the complexities of model configuration, dependency management, and resource optimization . It functions analogously to Docker for AI models, allowing developers to "pull" pre-packaged models and run them with a simple command-line interface (CLI) or a programmatic REST API. This approach is particularly well-suited for the project's constraints, which include limited hardware capabilities and a strict requirement for local, private operation. By leveraging Ollama, the system can efficiently run models like TinyLlama and, if resources permit, larger models such as GPT-OSS:20b, without relying on external cloud services, thereby ensuring data privacy and minimizing operational costs . The following sections detail the architecture for integrating Ollama into the containerized Docker environment and the methods for programmatic interaction from other services.

### 6.1 Ollama as the Model Management Engine

Ollama serves as the foundational layer for all LLM-related operations within the system architecture. Its primary role is to act as a self-contained service that manages the entire lifecycle of local language models, from initial download and setup to serving inference requests. This centralized management is critical for maintaining a clean separation of concerns, where other application components, such as the chatbot or research tool, do not need to handle the intricate details of model execution. Instead, they communicate with Ollama through a standardized API, requesting text generation or chat completions. This design not only simplifies the development of downstream services but also enhances the system's modularity and scalability. For instance, if a new, more powerful model becomes available, it can be integrated into the system simply by pulling it via Ollama's CLI and updating the model name in the API calls of the respective services, without requiring any changes to their core logic. The Ollama engine is optimized for performance, utilizing libraries like `llama.cpp` under the hood to ensure efficient execution on both CPU and GPU hardware, which is a crucial feature for achieving acceptable performance on resource-constrained local machines .

#### 6.1.1 Running Ollama in a Docker Container

To align with the project's containerized deployment strategy, Ollama itself will be encapsulated within a dedicated Docker container. This approach ensures that the Ollama service and its dependencies are isolated from the host system and other application services, promoting consistency and reproducibility across different development and deployment environments. The `docker-compose.yml` file will define an `ollama` service, specifying the official Ollama Docker image (e.g., `ollama/ollama`). This container will expose the necessary port (by default, `11434`) to allow other containers in the same Docker network to communicate with the Ollama API. A crucial aspect of this setup is the management of model data. Since LLMs can be several gigabytes in size, it is inefficient and impractical to include them in the Docker image itself. Instead, a Docker volume will be mounted to a directory within the container where Ollama stores its models (e.g., `/root/.ollama`). This volume ensures that downloaded models are persisted on the host machine's filesystem, preventing the need to re-download them every time the Ollama container is restarted or recreated. This strategy significantly speeds up the startup time of the system after the initial model pull and provides a mechanism for backing up or sharing the model cache.

#### 6.1.2 Pulling and Managing Models (TinyLlama, GPT-OSS:20b)

Ollama simplifies the process of acquiring and managing LLMs through its intuitive command-line interface. The system will be designed to work with a predefined set of models suitable for the different use cases. For the chatbot and general-purpose tasks, a lightweight yet capable model like \*\*TinyLlama\*\* is an ideal starting point, given the hardware constraints. For more complex tasks, such as in-depth research or sophisticated code generation, the system can be configured to use a larger model like \*\*GPT-OSS:20b\*\*, provided the local hardware has sufficient RAM and computational power. The process of making these models available to the system is straightforward. Upon the first startup, or as part of an initialization script, the system can execute commands like `ollama pull tinyllama` and `ollama pull gpt-oss:20b`. These commands connect to the Ollama model library, download the specified model weights and configurations, and store them in the mounted volume. The `docker-compose.yml` can be configured to run an initialization container that executes these pull commands, ensuring that the required models are available before the main application services start. This automated setup process makes the system easy to deploy and configure for different hardware capabilities, allowing users to select the models that best fit their needs and resources.

#### 6.1.3 Exposing the Ollama API for Service Communication

Once the Ollama service is running within its Docker container, it exposes a REST API that serves as the primary interface for all other services to interact with the managed LLMs. This API provides endpoints for various operations, including generating text, engaging in chat conversations, and listing available models. By default, the API is accessible at `http://localhost:11434`. In a Docker Compose environment, other services can reach the Ollama container using its service name as the hostname (e.g., `http://ollama:11434`). This service discovery mechanism is a key feature of Docker Compose, simplifying inter-service communication. The API is designed to be simple and intuitive. For example, a `POST` request to the `/api/generate` endpoint with a JSON payload containing the model name and a prompt will return the model's generated text. Similarly, the `/api/chat` endpoint is used for conversational interactions, accepting a list of messages and returning the model's response. This standardized, HTTP-based API allows services written in different programming languages to easily integrate with the LLM engine. The Python-based services in this MVP, for instance, will use the official Ollama Python library, which is a thin wrapper around this REST API, providing a more convenient and Pythonic interface for making requests and handling responses .

### 6.2 Interacting with Ollama via Python

The backend services for the chatbot, code assistant, and research tool will be primarily developed in Python, leveraging its rich ecosystem of libraries for AI and web development. To facilitate seamless communication with the Ollama engine, the official `ollama` Python library will be used. This library provides a clean, high-level interface for interacting with the Ollama REST API, abstracting away the low-level details of making HTTP requests and parsing JSON responses. It allows developers to integrate powerful LLM capabilities into their applications with just a few lines of code, making it an ideal choice for rapidly developing the MVP components . The library supports all the features of the Ollama API, including text generation, chat, streaming responses, and even multi-modal inputs for models that support them, such as LLaVA . This comprehensive feature set ensures that the Python-based services can fully leverage the capabilities of the underlying LLMs managed by Ollama. The following subsections explore the specific methods for integrating Ollama into the Python-based services, from basic API calls to more advanced agentic behaviors.

#### 6.2.1 Using the Official Ollama Python Library

The `ollama` Python library is the recommended and most straightforward way to integrate local LLMs into Python applications. Installation is simple, requiring only a `pip install ollama` command . Once installed, it provides a set of intuitive functions that mirror the functionality of the Ollama CLI and REST API. For instance, the `ollama.generate()` function can be used for single-turn text completion tasks. A developer simply needs to provide the model name and a prompt string, and the function will return the model's response. For more interactive, conversational use cases, such as the chatbot, the `ollama.chat()` function is used. This function takes a model name and a list of messages, where each message is a dictionary with a `role` (e.g., "user", "assistant", "system") and `content`. This allows the application to maintain a conversation history and provide context to the model, which is essential for coherent and context-aware interactions . The library also supports streaming responses by setting the `stream=True` parameter, which is crucial for creating a responsive user interface where the model's output appears in real-time, token by token, rather than being returned all at once after a potentially long delay. This feature significantly improves the perceived performance and user experience of the chatbot and other interactive tools.

A key feature of the Ollama Python library is its support for function calling, which enables the creation of more sophisticated AI agents. This allows an LLM to determine when to call a specific Python function to retrieve information or perform an action, extending its capabilities beyond its internal knowledge. For example, a code assistant could be given a function to execute Python code, or a research tool could be given a function to search a local document database. The library handles the process of making the function's signature and documentation available to the model and then executing the function with the arguments provided by the model in its response . This powerful feature is central to building the advanced use cases required for this MVP, transforming the LLM from a simple text generator into an active agent that can interact with its environment and perform complex tasks.

#### 6.2.2 Making Requests to the Ollama REST API

While the official Python library is the most convenient method, it is also possible to interact with Ollama by making direct HTTP requests to its REST API. This approach offers more flexibility and can be useful in scenarios where the Python library is not available or when integrating with services written in other languages. The API is well-documented and follows standard REST conventions. For example, to generate text, a `POST` request is sent to `http://<ollama-host>:11434/api/generate` with a JSON body containing the `model` and `prompt`. The response is a stream of JSON objects, each representing a token of the generated text. For chat interactions, the endpoint is `/api/chat`, and the request body includes the `model` and a `messages` array. This direct API access can be implemented using any HTTP client library, such as Python's `requests` library. This method provides a lower-level control over the interaction, allowing developers to handle the raw response stream and implement custom logic for processing the model's output. It also serves as the underlying mechanism for the official Python and JavaScript libraries, ensuring that any client that can make HTTP requests can leverage the power of Ollama's local LLMs .

#### 6.2.3 Implementing Function Calling for Advanced Agentic Behavior

The function calling capability of the Ollama Python library is a cornerstone for building the advanced features of the code assistant and research tool. This feature allows the LLM to act as a reasoning engine that can decide to use external tools to accomplish a user's request. The implementation involves several steps. First, the developer defines a standard Python function that performs a specific task. This function should have a clear name, type hints for its parameters, and a descriptive docstring, as this information is used by the library to generate a JSON schema that helps the LLM understand the function's purpose and how to use it . For example, a function to add two numbers might look like this:

```python

def add\_two\_numbers(a: int, b: int) -> int:

"""

Add two numbers and return the result.

"""

return a + b

```

Next, this function is passed to the `ollama.chat()` call via the `tools` parameter. When the user asks a question like "What is 10 + 10?", the LLM can analyze the request and decide to call the `add\_two\_numbers` function instead of trying to compute the answer itself. The Ollama library then parses the model's response, extracts the function name and arguments, and executes the corresponding Python function. The result of this function call can then be fed back into the conversation, allowing the LLM to formulate a final answer for the user . This mechanism can be extended to more complex scenarios. For the code assistant, a function could be defined to execute a block of code in a sandboxed environment and return the output. For the research tool, a function could query a vector database to retrieve relevant documents based on a search query. This ability to chain together LLM reasoning with external tool execution is what enables the creation of truly powerful and autonomous AI agents, all running locally and privately within the Dockerized environment.

## 7. Data Storage and Management

A robust data storage and management strategy is essential for the MVP to function correctly and provide a seamless user experience. The system needs to store user data, such as chat histories and session information, as well as the knowledge base for the research tool, which consists of vector embeddings of the ingested documents. The architecture is designed to use lightweight and efficient storage solutions that are well-suited for a local deployment environment.

### 7.1 Persistent Storage for User Data

To ensure that user data is not lost when the containers are stopped or restarted, the system will use persistent storage. This will be achieved through the use of Docker volumes, which allow data to be stored on the host machine's filesystem and mounted into the containers. The system will use a lightweight database to store user data, such as chat histories and session information.

#### 7.1.1 Storing Chat Histories and User Sessions

The chatbot and other interactive features of the system will store user data, such as chat histories and session information, to provide a personalized and context-aware experience. This data will be stored in a lightweight database, such as SQLite or PostgreSQL. The database will be run in a separate Docker container, and a volume will be mounted to persist the data on the host machine. This ensures that the user's data is not lost when the containers are restarted, and it also allows for the data to be backed up and restored.

#### 7.1.2 Using a Lightweight Database (e.g., SQLite, PostgreSQL)

For storing user data, the system will use a lightweight database like SQLite or PostgreSQL. SQLite is a serverless database that is easy to set up and use, making it a good choice for a simple MVP. PostgreSQL is a more powerful and feature-rich database that is well-suited for more complex applications. The choice of database will depend on the specific requirements of the system, but both are good options for a local deployment.

### 7.2 Knowledge Base for the Research Tool

The research tool requires a knowledge base to store the vector embeddings of the ingested documents. This knowledge base will be used to retrieve relevant information in response to user queries. The system will use a vector database to store the vector embeddings, as it is a specialized database that is optimized for storing and querying high-dimensional vectors.

#### 7.2.1 Storing Vector Embeddings in a Vector Database

The research tool will use a vector database to store the vector embeddings of the ingested documents. A vector database is a specialized database that is optimized for storing and querying high-dimensional vectors. The system will use a lightweight and efficient vector database like Faiss or ChromaDB. The vector database will be run in a separate Docker container, and a volume will be mounted to persist the data on the host machine.

#### 7.2.2 Managing Document Collections and Metadata

In addition to storing the vector embeddings, the research tool will also need to manage the document collections and their metadata. This includes information such as the document's title, author, and date of creation. This metadata will be stored in a separate database, such as a relational database or a document database. The metadata will be used to provide additional context for the retrieved information, and it will also be used to filter the search results.