**Local RAG – Retrieval-Augmented Generation Project Documentation**

*Graduation Project – Comprehensive Technical Report*

**Author:** *YourNameYour Name*  
**Supervisor:** *SupervisorNameSupervisor Name*  
**Institution:** *University/DepartmentUniversity/Department*  
**Date:** *June 2025*

**Table of Contents**

1. **Introduction**  
   1.1. Background and Context  
   1.2. Problem Statement  
   1.3. Overview of Solution (Local RAG System)
2. **Theoretical Background**  
   2.1. Retrieval-Augmented Generation (RAG) – Concept and Significance  
   2.2. Embeddings and Similarity Search  
   2.3. Transformers and their Role in RAG Systems  
   2.4. Vector Databases – Overview and Importance
3. **Project Objectives and Motivation**  
   3.1. Project Goal and Scope  
   3.2. Motivation for a Local RAG Solution  
   3.3. Expected Outcomes and Use Cases
4. **System Architecture**  
   4.1. High-Level Architecture of the RAG System  
   4.2. RAG Pipeline Workflow  
   4.3. Document Storage with PostgreSQL and pgvector Extension  
   4.4. Embedding Generation with Cohere API  
   4.5. LLM Integration (Google PaLM API and DeepSeek Models)
5. **Technology Stack and Tools**  
   5.1. FastAPI (Backend Web Framework)  
   5.2. PostgreSQL Database and pgvector Extension  
   5.3. Cohere Embeddings API  
   5.4. Google PaLM API / DeepSeek LLM for Query Answering  
   5.5. Frontend Framework (Web Interface)  
   5.6. Docker and Deployment Configuration
6. **Codebase Structure and Module Walkthrough**  
   6.1. Directory Layout Overview (/src, /frontend, /docker, etc.)  
   6.2. Models and Schemas (/src/models)  
   6.3. API Routes and Controllers (/src/routes)  
   6.4. Services and Utilities (Embedding, Database, LLM factories)  
   6.5. Frontend Module Overview (/frontend)
7. **Main Processes – Pseudocode and Explanations**  
   7.1. Document Ingestion Pipeline (Data Loading and Processing)  
   7.2. Vector Generation and Storage Process  
   7.3. Query Processing and Answer Generation Process
8. **Project Planning and Management**  
   8.1. Development Timeline and Milestones  
   8.2. Team Roles and Contributions  
   8.3. Challenges Encountered and Solutions Implemented
9. **Future Work and Potential Improvements**  
   9.1. Enhancing Retrieval and Ranking  
   9.2. Scaling the System (Performance and Volume)  
   9.3. Extended Multi-modal RAG and Other Improvements
10. **Conclusion**
11. **Appendices**  
    A. Glossary of Terms  
    B. List of Figures and Tables  
    C. Bibliography (References)

**1. Introduction**

**1.1 Background and Context:** In recent years, large language models (LLMs) have demonstrated unprecedented capabilities in natural language understanding and generation. These models, based on the Transformer architecture, can generate human-like responses and have been applied to tasks ranging from chatbots to content summarization. However, out-of-the-box LLMs operate purely on knowledge contained in their training data, which may be static or limited. This poses challenges when up-to-date or specific domain knowledge is required from an AI system. To address this, the technique of *Retrieval-Augmented Generation (RAG)* has emerged as a powerful approach. **Retrieval-Augmented Generation** is a method of enhancing the accuracy and reliability of generative AI models by grounding their responses in external data sources. In essence, RAG systems retrieve relevant documents or facts from a knowledge repository and supply this context to the LLM at query time, so that the model’s generated answer is augmented with factual, relevant information from those documents. This approach effectively combines traditional information retrieval with modern text generation, thereby reducing issues like hallucinations (fabricated answers) and improving the specificity of responses. Many industry and academic solutions have adopted RAG as a key strategy, making it one of the most popular LLM-based system architectures by 2023.

**1.2 Problem Statement:** Despite their prowess, LLMs alone are insufficient for tasks that require detailed knowledge of specific, possibly proprietary, content. For example, a company might want a chatbot that can answer questions about its internal documentation or a student might wish to query a personal library of research papers. Using a remote API-only LLM (such as OpenAI’s GPT) in such cases can be problematic for reasons of privacy (data must be sent to a third-party) and accuracy (the model may not know recent or niche information). The problem this project addresses is: *How can we build a* ***local*** *question-answering system that leverages a user’s own documents to provide accurate, context-aware answers?* This breaks down into sub-problems: (a) storing and indexing a large collection of unstructured documents in a way that allows efficient semantic search, (b) generating high-quality embeddings (vector representations) of text so that semantic similarity can be computed, and (c) integrating an LLM that can utilize retrieved information to formulate answers. Additionally, the solution must be practical for end-users (with an interface for queries and document uploads) and should be designed in a *production-ready* manner (robust, containerized, and maintainable).

**1.3 Overview of Solution (Local RAG System):** The proposed solution is a system called **Local RAG**, which implements a Retrieval-Augmented Generation pipeline entirely on the user's own infrastructure (or a controlled environment). The core idea is to ingest the user’s documents into a local database with vector search capabilities, and to use an LLM to answer questions by referring to those documents. The system architecture relies on a combination of **FastAPI** (a high-performance Python web framework) for the backend API, **PostgreSQL** (a relational database) with the **pgvector** extension for vector similarity search, and external AI services for language processing – specifically **Cohere’s embedding API** for converting text to vectors and either **Google’s PaLM API or an open-source DeepSeek LLM** for generating answers to queries. A front-end web interface allows users to upload documents and ask questions, which are handled by the FastAPI backend. The expected workflow is that when a user asks a question, the system will find the most relevant document pieces from the database (using vector similarity search on embeddings) and then feed those to the LLM so it can produce a well-informed answer. The emphasis is on making the system *local* or self-hosted as much as possible, meaning the vector database and potentially even the LLM can run in the user’s environment (for example, via a local LLM server) to preserve data privacy. This documentation will delve into the theoretical foundations of such a system (Chapter 2), detail the specific architecture and components of the implemented Local RAG project (Chapters 4-6), outline the development process and project management aspects (Chapter 8), and discuss future improvements (Chapter 9). In summary, Local RAG is a minimal yet production-oriented implementation of a RAG-based question answering system, aimed at demonstrating how private data can be coupled with cutting-edge language models to deliver accurate, context-rich answers.

**2. Theoretical Background**

This chapter provides the foundation for understanding the Local RAG system by covering key concepts and technologies that underpin Retrieval-Augmented Generation. We will discuss what RAG entails and why it’s useful, explain embeddings and how similarity search works, give an overview of Transformers (the neural network architecture behind modern LLMs) and describe vector databases and their role in such systems. This theoretical grounding will clarify *why* certain design choices were made in the project’s implementation.

**2.1 Retrieval-Augmented Generation (RAG) – Concept and Significance**

Retrieval-Augmented Generation (RAG) refers to AI systems that combine a *retrieval* step with a *generation* step to produce answers or content. In a RAG system, a query (e.g. a user’s question) is first used to **retrieve** relevant information from an external knowledge source (such as a document corpus or database). This retrieved context is then provided to a **generative model** (typically an LLM) which **augments** its answer with the provided information. The motivation behind RAG is to overcome limitations of standalone language models: LLMs have finite and static knowledge (up to their training cutoff) and can sometimes generate plausible-sounding but incorrect answers (hallucinations). By grounding the generation in real data, RAG improves the **accuracy and reliability** of responses.

In practical terms, RAG can be seen as *open-book question answering*, where the model has a “book” (the document repository) to refer to, as opposed to *closed-book* where it must rely on memorized training data. This concept was formalized by Lewis et al. (2020) in their work on knowledge-intensive NLP, which coined the term RAG for such architectures. Many commercial QA systems and chatbots use RAG techniques. For example, search engines with AI chat (like Bing Chat or Perplexity.ai) will fetch web results and then have an LLM formulate an answer citing those results. RAG has thus become a dominant pattern for building *domain-specific assistants*, enterprise chatbots, and any scenario where the latest information or proprietary data needs to be incorporated into LLM outputs.

A RAG pipeline involves two main phases: an **offline indexing phase** and an **online query phase**. During indexing, the knowledge source (documents) is processed, broken into chunks, and indexed in a way that makes retrieval efficient (often via embedding vectors, see Section 2.2). During the query (online) phase, the system goes through *retrieve-and-read*: it first converts the user query into an internal form (like an embedding) to find relevant pieces of text, then constructs a prompt containing the question and those retrieved pieces, which is finally fed to the LLM to generate an answer. In essence, RAG = *Retrieval (of knowledge) + Generation (of answer)*.

The significance of RAG lies in multiple advantages it brings: (1) **Improved accuracy** – the LLM’s answer is grounded in factual information from the retrieval step, reducing guesswork. (2) **Up-to-date information** – even if the LLM’s training data is outdated, the retrieval step can fetch current data (for instance, latest documents or real-time database entries). (3) **Domain specificity** – RAG allows an LLM to operate on specialized corpora (legal documents, medical texts, company manuals, etc.) without requiring expensive fine-tuning of the model on those corpora. (4) **Source attribution and transparency** – because RAG systems know which documents contributed to an answer, they can provide citations or references, increasing user trust in the results. These benefits make RAG a compelling approach for building reliable AI assistants.

In summary, RAG leverages the strengths of both **retrieval systems** (which excel at finding relevant text from large collections) and **generative models** (which excel at language fluency and reasoning). Throughout this document, RAG will be the central paradigm around which our system is built. The following sections elaborate on the components that make RAG possible: embeddings for semantic search, Transformer-based LLMs, and vector databases for efficient retrieval.

**2.2 Embeddings and Similarity Search**

At the heart of the retrieval step in a RAG pipeline is the concept of **embeddings**. An *embedding* is a numerical representation of a piece of information (text, image, etc.) in a high-dimensional vector space. For text, an embedding is typically a list of floating-point numbers (often a few hundred dimensions long) that encodes the semantic meaning of the text. The key idea is that semantically similar texts will have embeddings that are *close* to each other in this vector space.

Modern AI models (often neural networks) are used to generate embeddings. Large language models or related neural encoders like BERT, GPT, or Cohere’s models take in a chunk of text and output an embedding vector. For example, a sentence about “economic growth” and another about “increase in GDP” might end up with vectors that are nearby in the space, whereas a sentence about “climate change” would be distant. These embeddings capture nuanced semantic and contextual information about the text. An important property of a well-trained embedding space is that **distance correlates with semantic similarity**: items with similar meaning cluster together, while dissimilar items are far apart. This allows us to use mathematical distance (like cosine similarity or Euclidean distance) as a proxy for “meaning” difference.

**Similarity Search** (also known as *semantic search*) is the process of finding items in a database whose embeddings are closest to the embedding of a given query. The flow is typically: when a user poses a query (in natural language), the system first computes the embedding of that query using the same embedding model used for documents. Then it compares this query embedding to all document embeddings in the repository to find the most similar ones (i.e., those with smallest distance). Only the top *K* results (K being a small number like 3 or 5) are returned as relevant contexts. This approach goes beyond simple keyword matching; it can find relevant text even if it uses different words than the query, as long as the meaning is related. For instance, a search for “finance report Q4 profits” might retrieve a document snippet talking about “fourth quarter earnings” because the embeddings of those texts would be close, even though the exact words differ.

To make this concrete: suppose we have an embedding vector for the query *“What are the health benefits of green tea?”*. The system will calculate this vector (say using Cohere’s embed API). Then it will search the vector database for vectors (representing document sentences or paragraphs) that have a high cosine similarity with the query vector. Perhaps it finds an article snippet “Green tea contains antioxidants that improve cardiovascular health.” Because the embedding of that snippet is close to that of the query, it’s deemed relevant. This snippet would then be retrieved for the generation phase.

Mathematically, if **q** is the query vector and we have document vectors **d₁, d₂, ..., dₙ**, we want to find those **dᵢ** that maximize similarity sim(q, dᵢ) (or minimize distance). Cosine similarity is a popular choice, defined as sim(q,d) = (q · d) / (||q||||d||). Many vector databases index vectors in a way that accelerates this nearest-neighbor search so that even with millions of vectors, one can efficiently retrieve the top-K similar ones.

In summary, **embeddings and similarity search enable semantic matching**: they allow the system to “understand” text beyond literal keywords, by working in a vector space of meanings. Our project uses this capability extensively — we embed all documents and store those embeddings in a database, and at query time we embed the user’s question and perform a similarity search to fetch relevant content. The quality of the whole RAG pipeline heavily depends on the quality of embeddings (they must capture domain-specific semantics well) and the efficiency of similarity search. The next section will briefly discuss the models that produce these embeddings (Transformers), and Section 2.4 will describe the database technology we use to store and search vectors.

**2.3 Transformers and Their Role in RAG Systems**

**Transformers** are a type of neural network architecture that has revolutionized natural language processing. Introduced by Vaswani et al. (2017) as the architecture behind models like BERT and GPT, Transformers use a mechanism called *self-attention* to process input text. In simple terms, **self-attention** allows the model to weigh the importance of different words in a sentence relative to each other when computing an internal representation. This means a Transformer can capture context—e.g., in the sentence “Tea has many health benefits, according to recent studies,” the model can learn that “health benefits” is an important phrase and relates to “tea” in a specific way.

Transformers are the backbone of both the embedding generators and the language model in a RAG system. For embeddings, often a Transformer-based model (like a pretrained BERT or a specialized embedding model such as Cohere’s Embed model) is used. It takes a text as input and outputs a fixed-size vector (the embedding). The Transformer’s ability to capture semantic relationships means these vectors are very informative about the text’s meaning. For generation, the LLM itself is typically a Transformer (e.g., GPT-3, PaLM, or open-source models like DeepSeek LLM). These models usually have an encoder-decoder or decoder-only Transformer architecture that allows them to take an input context (the prompt) and generate a continuation (the answer). The **decoder Transformer** attends to the entirety of the provided prompt (which in a RAG scenario includes the query and retrieved documents) and uses that information to produce a coherent answer.

In our Local RAG project, Transformers appear in two places:

* **Embedding model:** We use Cohere’s embedding API, which under the hood uses a Transformer-based language model trained to produce embeddings. (Cohere’s models are trained on vast text data to position semantically similar text near each other in vector space).
* **LLM for generation:** We interface with either Google’s **PaLM** (Pathways Language Model) via its API or an open-source **DeepSeek LLM** running locally. PaLM is a Transformer-based model developed by Google, and DeepSeek LLM is an open-source Transformer model (available in sizes like 7B and 67B parameters) noted for strong performance in reasoning and coding. Both are advanced generative Transformers.

The role of Transformers in the RAG system can be summarized as enabling the system to *understand and generate language*. The embedding model’s Transformer encoder “understands” the content of documents and queries enough to produce meaningful vector representations. The LLM’s Transformer decoder then “understands” the question in context of retrieved info and crafts a fluent answer. The self-attention mechanism in Transformers is critical here: it allows the model to integrate the retrieved facts into the answer by focusing on the most relevant parts of the prompt. For instance, if the prompt to the LLM is:

**User question:** “What are the benefits of green tea?”  
**Context document:** “...Green tea contains catechins, which are antioxidants that may improve metabolism and reduce risk of certain diseases...”,

the Transformer in the LLM will attend to the parts of the context that mention “benefits” and “green tea” when generating its response. Because of this capability, Transformers make it possible for RAG to yield answers that are not just accurate, but also contextually appropriate and well-integrated with the provided information. Essentially, the LLM (a Transformer) is prompted with both question and supporting text, and it binds the two: it uses the supporting text to ground its answer, effectively **binding retrieval with generation**. This synergy is why the Transformer-based architecture is so crucial and ubiquitous in RAG implementations.

**2.4 Overview of Vector Databases**

A **vector database** is a specialized data storage and search system designed to handle vector data – particularly high-dimensional embeddings. Unlike traditional relational databases (which store structured data in tables) or document stores (which store JSON or text), a vector database is optimized for **similarity search** on vectors. In a RAG system, after we convert documents into embeddings, we need a place to store those embeddings and query them efficiently. This is where vector databases come in.

A vector database can be thought of as an index of points in a high-dimensional space. It provides operations to insert new vectors (with an associated identifier or metadata, like which document and which part of document it came from), and to query the nearest neighbors to a given vector. Internally, these databases use data structures and algorithms (such as approximate nearest neighbor indices, e.g., HNSW or IVF) to make search fast even if there are millions of vectors.

There are several purpose-built vector database systems (often open-source or offered as managed services): examples include **Qdrant**, **Weaviate**, **Pinecone**, **Milvus**, **Chroma**, etc. All of these provide an API to do similarity search over stored embeddings. As an alternative, some traditional databases have added vector search capabilities through extensions or native types – PostgreSQL’s **pgvector** extension is one such solution. It allows Postgres to store vectors in a column and use an index to perform nearest-neighbor searches.

In this project, we utilize **PostgreSQL with pgvector** as our vector database. The decision to use Postgres/pgvector was influenced by practical considerations: PostgreSQL is an enterprise-grade, widely-used database, and adding pgvector means we can combine structured data (if needed) and vector search in one system. This avoids introducing an additional standalone vector DB server. The pgvector extension defines a new column type for vectors and provides indexing methods to perform approximate nearest neighbor search on those vectors. As a result, we can do a SQL query like:

SELECT id, content

FROM documents

ORDER BY embedding <-> query\_embedding

LIMIT 5;

Here <-> is an operator provided by pgvector for vector distance (for example, Euclidean distance or cosine distance if using normalized vectors). This query would retrieve the 5 documents with embeddings closest to query\_embedding. By integrating this into our application, we achieve semantic search within Postgres itself.

A **formal definition**: *“A vector database is a specialized system designed to efficiently handle high-dimensional vector data. It excels at indexing, querying, and retrieving this data, enabling advanced analysis and similarity searches that traditional databases cannot easily perform.”*. Traditional databases struggle with similarity search because they are built for exact matches or range queries on scalar values, not distances in 100+ dimensional spaces. Vector DBs, on the other hand, are built from the ground up for this purpose, often supporting metadata filtering (find similar vectors among those that match some criteria), hybrid queries (combining vector similarity with keyword filters), and CRUD operations for vectors.

In our Local RAG system, the vector database component is responsible for storing all document embeddings and retrieving relevant ones for any new query. The usage of Postgres/pgvector also aligns with ease of deployment – since we already use Postgres for other data (like storing the original documents or user info), extending it to vectors simplifies the architecture. Moreover, pgvector’s performance is quite solid and any small differences in speed compared to a specialized vector DB are usually negligible compared to the overall LLM response time. This means we get the benefit of vector search without a significant performance penalty, all while using familiar SQL tools.

To summarize this theoretical background: RAG systems stand on three pillars – retrieving relevant info (enabled by embeddings and vector databases), and generating answers (enabled by Transformer-based LLMs). In the chapters that follow, we will see how these concepts are implemented in the Local RAG project. But first, we will clearly articulate the objectives of the project and our motivation for undertaking it, in the next chapter.

**3. Project Objectives and Motivation**

**3.1 Project Goal and Scope**

The primary objective of this project is to **develop a full-stack, production-ready application** that demonstrates Retrieval-Augmented Generation on a local corpus of documents. In more concrete terms, the goal is to build a system where a user can upload a set of documents (PDFs, text files, etc.), and then ask questions in natural language to obtain answers that are grounded in the content of those uploaded documents. The system is expected to handle the entire pipeline: from document ingestion and indexing, through semantic search, to answer generation via an LLM. Importantly, the project focuses on the **“local”** aspect: the core data (documents and their embeddings) and possibly the model inference are handled on the user’s side or within a controlled environment, rather than relying entirely on cloud services. This implies considerations for data privacy and offline capabilities.

The scope of the project includes:

* Implementing a backend API service (using FastAPI in Python) to support uploading documents, querying, and retrieving answers.
* Setting up a persistent storage for documents and embeddings using PostgreSQL (with the pgvector extension for vector similarity indexing).
* Integrating with Cohere’s API to generate embeddings for pieces of text (documents and queries) – these embeddings are crucial for our semantic search.
* Integrating with at least one Large Language Model to generate answers. In our case, integration options are provided for a cloud-based model (Google’s PaLM API, representing a powerful closed-source LLM) and a local or open-source model (DeepSeek LLM, which can be run via a local server such as Ollama or similar).
* Developing a simple front-end (web-based user interface) to allow user interactions: file uploads and question asking, with display of answers (and possibly the context used for the answer, to enhance transparency).
* Containerization and orchestration using Docker, to allow the entire application stack (database, backend, etc.) to be run easily in different environments (development, testing, deployment).

The project’s **success criteria** are: a working system where a user can get accurate answers from their documents. The answers should be factually correct with respect to the source content (as far as the system’s capability goes), and the system should handle multi-document corpora and different query phrasing robustly. Performance (speed) is also a consideration: the system should retrieve and answer typical queries within a few seconds at most, which is feasible given the use of vector indexes and fast LLM APIs.

It’s worth noting what is *not* in scope: building a new embedding model or LLM from scratch (we leverage existing APIs/models), deeply optimizing at the C++ level the vector search (we rely on pgvector’s internal optimizations), or handling multimodal data like images (we focus on text documents). These could be areas of future extension but are outside the current scope.

**3.2 Motivation for a Local RAG Solution**

The motivation behind this project stems from both the limitations observed in standalone LLM deployments and a practical need for *privacy-preserving, domain-specific AI assistants*. Large Language Models like GPT-3 or PaLM are incredibly powerful, but without additional context they might give generic or incorrect answers for specialized questions. Additionally, sending sensitive documents to third-party AI services raises confidentiality concerns. Thus, the idea of *Local RAG* is appealing: can we equip a locally-run (or self-hosted) system with intelligence comparable to cloud AI, using our own data?

Several factors drove the motivation:

* **Bridging Knowledge Gaps:** As an example, if one needs an AI assistant to answer questions about a company’s internal policies or a researcher’s specific papers, a vanilla LLM often doesn’t have that information. RAG enables bridging that gap by feeding the model the information from those documents. We were motivated to implement such bridging in a reusable application.
* **Reducing Hallucinations:** We observed that models like ChatGPT sometimes “make up” answers when they aren’t sure. By using RAG, the model has factual text to refer to, which significantly reduces the incidence of fabricated answers. This was a strong motivator: we wanted more **trustworthy AI outputs**. It aligns with the idea of building AI that can cite sources and back up its statements with evidence (like our system potentially showing snippets from documents that support its answer).
* **Data Privacy and Local Control:** Many organizations (and individuals) are uncomfortable with uploading proprietary documents to an external service. By building the system to run with a local database and even potentially a local LLM (DeepSeek LLM is open-source and can be run on local hardware with sufficient resources), we cater to scenarios where data must remain on-premises. The motivation here is enabling AI on *private data* without compromise. Even when using a cloud LLM API (like Google’s) for generation, our design can ensure only minimal necessary context is sent over (and even that could be encrypted or redacted if needed).
* **Educational Value:** From a learning perspective (this being a graduation project), implementing a RAG system brings together various skills: database management, natural language processing, API integration, and full-stack development. The project was motivated by a desire to gain hands-on experience with these and to create a reference implementation that others can learn from. The step-by-step educational course (on which this project is based) was delivered in Arabic to help disseminate practical AI knowledge in that community.
* **Emerging Trend and Relevance:** RAG is at the forefront of many AI applications today. By working on this project, we aimed to contribute to this rapidly evolving area and keep ourselves at the cutting edge. Notably, major cloud providers (like Google in their Vertex AI Search) and startups are offering RAG solutions, which validates the importance of mastering this technique. We saw developing Local RAG as not just an academic exercise, but something with real-world applicability – from building smarter chatbots, to assisting customer support with relevant knowledge, to personalized educational tutors that know a student’s own notes.

In short, the motivation was to create a system that is **intelligent, accurate, and respects data locality**. The aspiration is that a user of Local RAG feels like they have their own “ChatGPT” but tuned to their personal or organizational knowledge base, running under their control.

**3.3 Expected Outcomes and Use Cases**

The expected outcome of the project is a fully documented, working prototype of a Local RAG application (which this document itself accompanies). This includes the code (made available on GitHub) and this documentation, which serves as a comprehensive guide for understanding, using, and extending the project. Specifically, we expect:

* A FastAPI server that can accept file uploads (for ingestion) and queries (for Q&A), responding with answers.
* A PostgreSQL database populated with the content of uploaded files, including a table of vector embeddings for fast search.
* The ability to switch between at least two LLM backends for answer generation (for example, use OpenAI/Google API in one mode, or a local LLM via an API in another mode).
* A front-end web interface (or at least a Postman collection / API documentation for demonstration) so that the system can be interacted with easily by an end-user.
* Realistic test cases demonstrating that, for instance, if a user uploads a document about “Green Tea Health Benefits” and asks *“Does green tea help with metabolism?”*, the system will answer with something like “Yes, green tea contains catechins which have been shown to boost metabolism” – ideally citing the relevant document snippet.

The use cases envisioned include:

* **Personal Knowledge Assistant:** A student or researcher feeding their notes and papers into the system and querying it in natural language to study or recall information.
* **Enterprise FAQ Bot:** A company ingests its internal wikis, manuals, and policy documents, then employees can query it to get quick answers (e.g., “How do I file an IT support ticket?” or “What is our policy on remote work?”).
* **Customer Support Assistant:** Feed in a product’s documentation and previous Q&A logs; support agents (or even customers via a chatbot) can query it to troubleshoot issues or find relevant info.
* **Education and Training:** Course materials can be ingested, allowing students to ask questions. The system could serve as a tutor that always refers back to the provided course content for answers.
* **Legal Document Search:** Upload a set of legal contracts or regulations; the user can ask questions and get answers with references to specific clauses (valuable in compliance or legal research).

Given the modular design, the system could be adapted to various domains by simply changing the data input. For example, for a medical Q&A system, one could ingest medical textbooks or research articles and then have doctors or patients query it.

The success of the project will be measured by how well it can handle these scenarios: relevance of retrieved info, correctness of answers, clarity of responses, and the ease with which new data can be added and used. Ultimately, the project should demonstrate that building a **local, domain-specific AI assistant is feasible** by combining the right set of tools (database, embeddings, and LLMs) in a coherent pipeline.

In the next chapter, we will dive into the *system architecture*, explaining how all these pieces (theoretical concepts and motivated choices) come together in the actual implementation of the Local RAG application.

**4. System Architecture**

In this chapter, we detail the architecture of the Local RAG system, highlighting its components and how data flows through the system. We place special emphasis on the RAG pipeline (how retrieval and generation are orchestrated), the role of PostgreSQL with pgvector as our vector store, the usage of Cohere’s embedding service, and the integration of LLMs (Google’s or DeepSeek’s) for answer generation. We will break down the architecture into its sub-parts and illustrate how they interact.

**4.1 High-Level Architecture of the RAG System**

*Figure 4.1: High-level system architecture for the Local RAG application. The user interacts via a frontend or API (FastAPI backend). Uploaded documents are stored in a Postgres database (with text and vector embeddings via pgvector). When a user asks a question, the system retrieves relevant documents by comparing the query’s embedding to stored embeddings, and then uses an LLM (via API, e.g., OpenAI/Google or a local DeepSeek model) to generate an answer based on the question and retrieved context.*

At a birds-eye view, the system is organized into three tiers: the **Frontend**, the **Backend**, and the **Database/Model services**.

* The **Frontend** (UI) provides a user interface for interactions. In our project, this is a web interface (a simple single-page application served separately from the FastAPI backend). The user can use this interface to upload files and to submit queries. The front-end communicates with the backend via HTTP requests (for example, a file upload via a POST request, or a query via a GET/POST request).
* The **Backend** is built with FastAPI (Python). This layer contains the application logic and various API endpoints (routes). The backend is responsible for orchestrating the RAG process: when a file is uploaded, it processes and stores it; when a question is asked, it triggers the retrieval and then calls the language model to generate an answer. The backend code is modular, with separate submodules handling different concerns (routing, database access, embedding calls, LLM calls, etc., which we will discuss in Chapter 6). FastAPI also automatically generates interactive API docs (Swagger UI) because it’s documented with Pydantic models, aiding in testing the endpoints.
* The **Database/Model Services** layer includes:
  + **PostgreSQL with pgvector**: This is where documents and their vector embeddings are stored. We have a table (or set of tables) for documents (or document chunks) which includes a VECTOR type column (provided by pgvector) for the embedding. We also use Postgres to store any other metadata (e.g., document titles, original filenames, etc.). The pgvector extension allows us to create an *index* on the vector column to speed up similarity search. Essentially, Postgres here functions as our vector database.
  + **Cohere Embedding API**: Although not self-hosted, this is an external service we rely on for generating embeddings. Whenever we ingest a new document or receive a query, the backend sends a request to Cohere’s API with the text, and the API returns the embedding vector. We could replace this with an open-source embedding model in the future, but for this project Cohere was chosen for its ease of use and quality of embeddings.
  + **LLM Service**: This can be one of two (configurable):
    - *Google’s LLM (via PaLM API)*: Representing a powerful cloud-based model. The backend can call Google’s generative language API with the composed prompt (question + context) to get an answer.
    - *DeepSeek LLM (via Local Server)*: Representing an open-source model that can be run locally. The project’s tutorial series included using **Ollama** (a tool to run LLMs on local hardware) to serve models like LLaMA or DeepSeek. In our context, DeepSeek LLM is an advanced model (67B parameters) available for local deployment. The system can be configured to use a local endpoint (for example, http://localhost:port/generate) to get answers from such a model. This requires that the model weights are set up on the server running the application.

By keeping the LLM integration abstract (through what we call an **LLM Factory** in the code), we can swap in different models or providers without changing the rest of the code. This flexibility is key; for instance, during development one might use OpenAI’s GPT-4 API (with an API key), but in a deployed environment with strict privacy, one might switch to a local DeepSeek model. The *LLM Factory* design encapsulates these options.

The high-level data flow is as follows:

1. **Document Ingestion Path:** The user uploads a document through the frontend. The file is sent to the FastAPI backend. The backend receives the file and processes it – this might include reading text from it (if PDF or DOCX, using a parser), splitting the text into smaller chunks (paragraphs or sections, to have finer granularity for retrieval), and then for each chunk, calling the Cohere API to get an embedding. The text and its embedding (and perhaps a reference to which document and which part of it it is) are then stored in the Postgres database. This ingestion process may also create records in a documents table and a vectors table, or a combined table, depending on schema design. We also log or store any relevant metadata (upload time, user id if multi-user system, etc.). Once stored, those embeddings are indexed and ready to be searched.
2. **Query Answering Path:** The user enters a question in the frontend UI (e.g., “What is the main topic of Document X?”). The frontend sends this query to the backend (likely hitting a /query endpoint with the question text). The FastAPI backend then handles this in several steps:
   * It first calls the Cohere Embed API to get the embedding for the query (essentially turning the user’s question into a vector in the same semantic space as the documents).
   * With this query embedding, the backend performs a similarity search in Postgres: using a SQL query (or an SQLAlchemy query) that leverages pgvector’s <-> distance operator to find the top K most similar document chunks. These top results are fetched from the database; let’s call them the “candidate context passages”.
   * Next, the backend constructs a prompt for the LLM. Typically, this prompt might be something like: “*Here are some relevant excerpts from documents: [excerpt 1] ... [excerpt K]. Using this information, answer the question: [user’s question]?*”. The exact formatting can vary, and prompt engineering is an important aspect to get right (we want the LLM to understand that the excerpts are to be used as reference and that it should base its answer on them).
   * The backend then uses the LLM service. If using Google PaLM API, it sends the prompt to that API and waits for a response. If using a local model (DeepSeek via Ollama), it sends the prompt to the local server. In either case, it receives back a generated answer (a text string). This answer ideally contains the information from the documents, phrased in a coherent way.
   * Optionally, the system might also extract which document or source each piece of information came from and could attach source references to the answer. Our current implementation might not fully do citation generation, but it’s something considered (especially since our context passages have identifiable sources, we could append something like “[Source: Document X]” in the answer).
   * Finally, the answer is sent back to the frontend, which displays it to the user. The user sees the answer (and possibly can click to see sources, etc., depending on UI features).

Throughout this process, various checks and features ensure robust operation: for example, if the user’s query returns no relevant passages (vector similarity below a threshold), the system might respond with “I’m sorry, I couldn’t find relevant information in the documents.” Also, if the LLM returns an unhelpful response, we might handle that (though fine-tuning that is an ongoing challenge).

The above describes how components interact. Let’s now zoom into certain parts of the architecture with more detail: the RAG pipeline specifically (retrieval + generation steps), the data storage via Postgres/pgvector, the embedding generation, and the LLM integration.

**4.2 RAG Pipeline Workflow**

The RAG pipeline can be thought of in three stages: **Ingestion/Indexing**, **Retrieval**, and **Generation**. We’ve outlined these in the context of architecture, but here we provide a stepwise description and highlight how data transforms at each step.

* **Ingestion & Indexing Phase:** *(Offline or pre-processing step)*
  + Input: Raw documents (could be PDFs, text files, etc.) possibly containing multiple pages or sections of text.
  + Processing:
    1. **Parsing**: Convert each document into plain text. (For PDF, we might extract text; for Word, perhaps use an .docx parser; for simple text, just read it).
    2. **Chunking**: Split the text into smaller chunks. The reason for chunking is twofold: (a) There is usually a limit to how much text we want to embed at once (embedding APIs often have limits on input size for efficiency, and embedding very large chunks might dilute the relevance), and (b) when retrieving, smaller chunks allow more precise matching to a query. We might split by paragraphs or headings or after N characters. In code, we may have a utility that splits text by sentences or newline, aiming for chunks of, say, 100-300 words.
    3. **Embedding**: For each chunk, call Cohere’s embed API to obtain an embedding vector. Each vector might be, for example, a 768-dimensional float array (the exact dimensionality depends on the model used by Cohere, e.g., embed-english-v2.0 might be 4096-d, etc., but we treat it abstractly). This step is crucial because it transforms text into the mathematical representation needed for similarity search.
    4. **Database Insert**: Store the chunk and its embedding in the database. The schema could have a table document\_chunks with columns like id, document\_id, chunk\_text, embedding\_vector. The embedding\_vector is of type vector (provided by pgvector). We also update any other related tables like a documents table which might store the original file info and link to chunks.
    5. **Indexing**: Ensure the pgvector index is updated (if using GIN or IVF indexes for approximate search). In Postgres, if we have already created an index on the embedding\_vector column, it updates automatically as we insert. If using approximate search (like HNSW index via pgvector), the index build might be an operation we run after bulk insertion. Given our incremental approach (upload one doc at a time), we likely use the “approximate vector search” feature with ivfflat index which can be updated as we go (with occasional reindexing if needed).
  + Output: The database now contains the document content in vectorized form, ready for retrieval. We can consider the knowledge “indexed”.
* **Retrieval Phase:** *(Online, triggered by a query)*
  + Input: A user’s query (text question).
  + Processing:
    1. **Query Embedding**: The query is sent to Cohere’s embed endpoint to produce an embedding vector in the same space as the documents. Let’s denote this vector as **q**.
    2. **Similarity Search**: The backend performs a nearest-neighbor search in the document\_chunks table where embedding\_vector is compared to **q**. In SQL, the order by clause might look like ORDER BY embedding\_vector <-> '[q]' LIMIT K (where [q] is the vector literal of the query, and <-> is the distance operator). K is pre-configured (commonly 3 or 5). The result is a set of top-K chunks, say C1,C2,...,CkC1, C2, ..., Ck, each with some similarity score. These are the “relevant contexts” that the system believes are related to the question.
    3. **Context assembly**: We extract the text of those top chunks. If needed, we might also fetch their document titles or IDs to possibly mention sources. The retrieved text may be concatenated or listed. There is a design choice: either feed all top-K chunks together to the LLM, or perhaps pick only the very top one if it’s clearly the best. Usually, using multiple chunks is helpful if information is spread out or if some chunks only partially match.
    4. **Prompt Construction**: We build the prompt for the generation phase. This often includes an instruction and the context. For example:  
       "Use the following information from the documents to answer the question.\n\nContext:\n[1] {C1}\n[2] {C2}\n...\nQuestion: {user\_question}\nAnswer:"  
       This prompt format is designed to clearly delineate what is context and what is the actual question, and to nudge the LLM into providing an answer drawing on the context. There are variations to prompt design. Some systems also instruct the LLM to give answers with references like “[1]” etc., if they want inline citations (this is an advanced prompting technique and may require a custom trained model or additional post-processing).
    5. **Selecting LLM**: Before moving to generation, the system selects which LLM to use. This could be based on configuration: e.g., an environment variable or a setting in the .env file might specify LLM\_PROVIDER=google vs LLM\_PROVIDER=local. The LLM Factory in code will, based on that setting, route the prompt to the appropriate interface. The Google PaLM integration might involve using Google’s AI Platform SDK or their REST API; the local DeepSeek via Ollama might involve sending an HTTP request to an Ollama server endpoint with the prompt.
  + Output: The stage yields a prepared prompt and selected context ready to send to the language model.
* **Generation Phase:** *(Online, immediately after retrieval)*
  + Input: The prompt containing context and question, delivered to the chosen LLM.
  + Processing:
    1. **LLM Processing**: The language model (Transformer-based, as discussed) receives the prompt and generates a response. Because we provided relevant context, the model’s task is simplified to focusing on that context to answer. For instance, if the context snippet says “Green tea contains catechins which boost metabolism”, and the question is about health benefits of green tea, the model will incorporate that fact into the answer. A well-phrased answer might be: “Green tea may improve metabolism due to the presence of catechins (antioxidants). Studies have shown it can reduce risk of certain diseases, contributing to its health benefits.”
    2. **Post-processing**: Once the raw answer text is obtained from the LLM, we might do some cleaning or formatting. For example, ensure it’s not ending mid-sentence (sometimes APIs return partial completion, so we ensure we got all tokens). Or if we requested citations and the model gave e.g. “[1]”, we might replace the numeric placeholders with actual document names using the mapping of our context docs. The extent of post-processing depends on how complex we made the prompt. In our initial implementation, we might keep it simple: just take the answer as-is.
    3. **Returning the Answer**: The backend sends the answer text back as the response to the query request. If using FastAPI, this could be a JSON response with fields like {"answer": "...", "sources": [...]} or just the answer string. Our design might include sources, e.g., we could attach the document titles of C1...Ck as a list of sources that were used. This is useful for transparency (and in an academic or enterprise setting, providing sources is often required). Given the system does have that info, including it is a bonus feature.
  + Output: The final answer to be delivered to the user.

This pipeline ensures that the LLM’s output is *augmented by retrieval*: the heavy lifting of finding relevant info is done by the similarity search, and the heavy lifting of composing a natural answer is done by the LLM. Each is used for what it’s best at.

To connect this with the earlier diagram (Figure 4.1): Steps in retrieval phase correspond to the arrow from the backend to Postgres and back (for vector search), and steps in generation phase correspond to the arrow from the backend to the LLM API and back. The figure illustrated a simple three-step sequence matching what we described: (1) user question goes to pdf-analyzer (our backend), (2) the service gets related documents from Postgres, (3) the service calls the LLM with question + docs to get an answer.

**4.3 Document Storage with PostgreSQL and pgvector Extension**

A crucial part of the system is the document store, which not only keeps the documents but also enables vector-based similarity search. We chose **PostgreSQL** as the database and leveraged the **pgvector** extension to handle vector data. Here we provide details on how that is set up and why it’s beneficial.

**Schema Design:** We created a schema that revolves around two main tables:

* documents – This table stores metadata about each uploaded document. For example: id (serial primary key), title (text), filename, upload\_date. It could also store a full text concatenation of content if needed for other purposes, but in our design we focus on storing text in chunks.
* chunks (or document\_chunks) – This table stores the chunks of text along with their embeddings. Key columns: id (serial PK), document\_id (foreign key referencing documents), content (text of the chunk), embedding (vector type). We also include maybe chunk\_index to indicate the order of the chunk in the original doc, which could be useful for reconstructing or for display context.

We applied the pgvector extension in the database (this requires installing the extension in Postgres, which is often done via a migration or a manual SQL command: CREATE EXTENSION IF NOT EXISTS vector;). The embedding column is then declared as type vector(768) if the embedding dimension is 768, for example. Actually, with pgvector, you specify the dimension (e.g., VECTOR(768) for a 768-dim vector). All embeddings inserted must be of this length or it will error.

**Indexing:** We create an index on the embedding column to accelerate similarity queries. For example: CREATE INDEX idx\_chunks\_embedding ON chunks USING ivfflat (embedding vector\_cosine\_ops) WITH (lists = 100);. Here ivfflat is one of the index types provided by pgvector for approximate search (which is faster for large data), and vector\_cosine\_ops specifies we want to use cosine distance as the measure (since our embeddings are likely normalized and cosine similarity is a good metric for semantic closeness). We can tune the number of lists (which affects recall/speed tradeoff). For small-scale usage, even a brute-force scan (which pgvector would do if no index or using the flat index type) is fine. But we prepared for scale by using an index.

**Why Postgres/pgvector:** Using PostgreSQL with pgvector provides a unified storage solution. Rather than having a separate vector DB and a separate relational DB for metadata, we have one system that can do both. As noted earlier, Postgres is widely adopted in enterprise settings, so this choice increases the chance that our project could be integrated or deployed in real-world scenarios without introducing a niche technology that the ops team is unfamiliar with. Additionally, Postgres offers transactional guarantees and flexibility of SQL. We can run complex queries, join with other data (for example, if we had user-specific document permissions, we could join with a permissions table to filter chunks by user before searching). This flexibility was a motivator.

**Alternatives considered:** Initially, the tutorial series (on which this project is based) introduced **MongoDB** for storing documents (steps 9-13 of the course used Mongo as a document store) and later introduced a separate vector DB (Qdrant) for similarity search (step 15 of the course). We experienced the complexity of maintaining two data stores. By step 20-21, the course itself switched to Postgres + pgvector, which simplified things. Our documentation focuses on the **tut-014 branch**, which was around the time of implementing the LLM Factory, but we incorporate knowledge from later improvements (since the final architecture includes Postgres/pgvector). Thus, in our project we have adopted that final architecture state. The rationale is clearly supported by community practices: combining Postgres and pgvector has become a popular approach for RAG because it leverages mature database infrastructure while adding vector search capabilities. EnterpriseDB notes that “While the combination of Postgres and pgvector enables us to utilize Postgres as a vector database, a complete AI application requires more” – indeed, we complement it with the rest of our pipeline.

**Data Persistence and Migrations:** We use SQLAlchemy (the Python ORM) and Alembic for migrations, as indicated in the repository. For example, after setting up the project, one of the steps is running alembic upgrade head to apply database migrations. These migrations set up the necessary tables and the pgvector extension. Using Alembic allows versioning the database schema changes (especially useful as the tutorial progressed from using Mongo to Postgres, migrations would capture adding new tables, etc.). In our code, we likely have SQLAlchemy models defined for Document and Chunk that correspond to the tables above, including appropriate field types.

**Document Text vs. Embeddings in DB:** One might wonder, do we store the actual text of each chunk in the database, or only the embeddings? In our implementation, we store both. The content text of each chunk is stored, which is needed to later feed into the LLM (we need actual text for that). Storing text in the DB is fine for moderate sizes (Postgres can handle text, and we can use TEXT type or even JSON if we had structured segments). If the text chunks are huge, it might be a consideration to store just embeddings and an identifier to an external text store, but in our case, storing in the DB is simplest.

**Capacity and Performance:** The design should scale to at least tens of thousands of chunks without trouble. Each embedding vector is perhaps a few kilobytes (e.g., 768 floats, if 4 bytes each, ~3KB per vector). 10k such vectors is ~30MB, which is trivial for Postgres. The similarity search with an index will be very fast for that size. Even up to millions of vectors, pgvector has been shown to work (though a dedicated vector DB might be faster for extremely large scale – but our project’s scope is not that large). Additionally, because our system operates possibly in a single-machine scenario (for a local deployment), using Postgres avoids having to run another service in parallel, which is pragmatic.

In summary, PostgreSQL with pgvector in our architecture serves as the **knowledge base** – it’s where all processed knowledge is kept. It provides the necessary operations to find relevant knowledge given a query.

**4.4 Embedding Generation with Cohere API**

Embeddings are the glue between user queries and documents, and we rely on **Cohere’s embedding service** to generate these. In our architecture, Cohere is an external component accessed via API calls from the backend. Let’s detail how and where this happens and why we chose Cohere.

**Cohere’s Embed Endpoint:** Cohere AI offers a simple REST API to get embeddings for text. The request typically includes the model name and the text (or texts) for which we want embeddings. For example, we might use a model like embed-english-light-v2.0 (just hypothetically) for generating 1024-dimensional embeddings. The API returns an array of floats for each input text. The documentation states: *“This endpoint returns text embeddings. An embedding is a list of floating point numbers that captures semantic information about the text it represents. Embeddings can be used to ... empower semantic search.”*. We use exactly this functionality: **embedding for semantic search**.

In implementation, we likely have a utility function or class (perhaps in src/services/embedding.py) that wraps Cohere’s API. For instance, using the cohere Python SDK, it might look like:

co = cohere.Client(api\_key=COHERE\_API\_KEY)

response = co.embed(texts=[text])

embedding = response.embeddings[0]

Or using raw HTTP requests to https://api.cohere.com/v2/embed with the necessary headers and JSON payload. Because our project is step-by-step built, at some earlier stage we might have been using OpenAI’s embeddings (given the README mentions setting OPENAI\_API\_KEY), but we’ll focus on Cohere as per the final emphasis. (It’s possible the user of this project switched to Cohere to avoid relying on OpenAI; however either could work since both provide embedding services. In any case, the system is abstract enough to allow either – one could configure which embedding provider, similar to LLM.)

**Where in the pipeline:** Embedding generation occurs:

* When ingesting a chunk of document text (as described, after chunking, for each chunk we call embed).
* When a query comes in (we call embed on the query string).

These calls need to be efficient and possibly batched. If a document has, say, 50 chunks, calling the API 50 times sequentially is slow due to overhead. Thankfully, Cohere’s API (and similarly OpenAI’s) allows batching: we can send an array of texts and get an array of embeddings in one API call. Our implementation likely does that – e.g., send maybe up to 5 or 10 chunks at a time in a batch to embed. This is a practical consideration for performance.

**Error Handling and Limits:** We ensure to handle API errors (like if the Cohere API key is invalid or quota exceeded). We also note the rate limits – if embedding many documents, we have to be mindful of not hitting requests/second limit. In a graduation project context, the volume is probably small, but we consider it.

**Choice of Cohere vs Alternatives:** The project specifically calls out Cohere, likely because:

* Cohere provides high-quality embeddings and has an academic-friendly vibe (their models are known to be good at semantic similarity).
* Possibly to avoid OpenAI dependency, as the user might not want to use OpenAI for cost or policy reasons.
* Cohere’s documentation on RAG (which we cited in section 2.1) shows they are promoting RAG usage, so their embed model is certainly viable for this use-case.

We could have also used OpenAI’s text-embedding-ada-002 model, which is a popular choice. That model yields 1536-d vectors and is state-of-the-art in many benchmarks for semantic search. If one sets the OPENAI\_API\_KEY, perhaps our code is flexible to use OpenAI’s embeddings (maybe a flag to choose provider). In any event, using either doesn’t change the architecture much – just which API endpoint is hit. For completeness, we document how Cohere is integrated since that was requested.

**Embedding Storage:** After receiving embeddings from Cohere, the backend typically converts them to a format suitable for Postgres insertion. If using an ORM, it might directly store a Python list/NumPy array to the vector field (some ORMs have support, or we might need to adapt). In raw SQL, we might convert the list to the Postgres array literal format or use a parameter binding. For example, using psycopg2, one can execute INSERT ... (embedding) VALUES (%s) and provide a Python list, and if psycopg2’s adapter is configured for pgvector, it stores it. In our case, we likely use SQLAlchemy + a pgvector plugin or a custom field type for vector. Indeed, there's a library sqlalchemy-pgvector that allows using a Vector type in SQLAlchemy models. Possibly the project used that for ease.

**Maintaining Consistency:** It’s important that the same model is used for both document and query embeddings. So, somewhere in config we have the model name or API settings. If Cohere releases a new model and one is tempted to switch, one must re-embed the documents to maintain the semantic space. We might mention that as a note: all embeddings in the database are produced by the same embedding model for consistency.

**Dimension and Data Type:** If Cohere’s model gives 768-dim vectors (as an example), our VECTOR(768) must match that. We know the dimension beforehand for a given model (Cohere API docs usually specify the output dim per model). If we mismatch, we’d get errors on insert. In our Alembic migration, we set the dimension correctly.

**Example:** Suppose a document chunk is “Green tea contains catechins which are antioxidants.” Cohere’s embed might produce a vector like [0.12, -0.03, ..., 0.07] (a 768-length list). For a query “Does green tea have antioxidants?”, the embedding might be [0.10, -0.01, ..., 0.08]. The cosine similarity of these two would be high, hence retrieval works. We rely on the embedding model to capture that “catechins which are antioxidants” relates to “antioxidants” in the question – something keyword search might miss if phrasing differs.

**Security & Privacy:** When using external APIs, one must consider data privacy. The content of documents and queries is being sent to Cohere (a third-party) for embedding. If our requirement is a fully local solution, this is a compromise. However, Cohere (and OpenAI etc.) typically have policies about not training on submitted data, etc., but still it's a leak. If needed, one could self-host an embedding model (there are open source embedding models such as sentence-transformers). That could be future work. For now, the convenience of the API was chosen. We assume the user of the system consents to that or is working with non-sensitive data.

To wrap up, embedding generation via Cohere is a straightforward but vital part of the pipeline. It transforms the data into the “language” that the vector database understands. Our implementation abstracts it enough so that if in future one swaps out Cohere for another embedding provider or local model, it would be easy (just modify the embedding service class). The result of this integration is that we have a semantic encoding of all texts, which as earlier described, powers the retrieval mechanism.

**4.5 LLM Integration (Google PaLM API and DeepSeek LLMs)**

The final piece of the architecture puzzle is how we integrate the **Large Language Model** for generating answers. The project mentions using **Google or DeepSeek LLMs** for answering queries, reflecting a flexible integration where either a cloud-based LLM (Google’s) or a local/open LLM (DeepSeek) can be used. We implemented this via an **LLM Factory** pattern – essentially a component that decides which LLM backend to use based on configuration and provides a unified interface to the rest of the system.

**Google’s LLM (PaLM API via GCP):** Google’s PaLM (Pathways Language Model) is accessible to developers through Google Cloud’s AI services (for example, Vertex AI or an API endpoint known as the Generative Language API). Using Google’s model likely requires API keys or service account credentials for GCP. The allure of using Google’s model is its high quality (comparable to GPT-3.5/GPT-4 depending on which version one gets access to), and maybe availability in certain regions or for certain languages. To integrate it:

* We would have to use Google’s SDK or REST calls. Google provides client libraries for Python to interact with their AI Platform. Possibly the project might use Google’s PaLM via an HTTP call. For instance, Google had an endpoint for their model (though details may require being part of their program).
* Another possibility: the user meant “Google” in a generic sense for any cloud LLM, but likely it specifically refers to PaLM (or maybe the LaMDA-based API they have).
* Configuration wise, we might have environment variables like USE\_GOOGLE\_LLM=True and GOOGLE\_API\_KEY or some credentials file.

When a query is processed and context gathered, if Google LLM is the target, we format a prompt and call the Google API. Google’s models usually expect either plain text prompt or a structured prompt (depending on if using their chat model). Since our use-case is Q&A, a plain prompt like we constructed would probably suffice. The response comes back, we parse out the answer text.

**DeepSeek LLM (Local model):** DeepSeek is an open-source LLM introduced recently (with 7B and 67B versions). To use it locally, we rely on a serving mechanism. The mini-rag course demonstrated using **Ollama**, which is a tool to run language models on macOS/Linux with GPU acceleration, etc., and exposes a local server where you can send prompts and get completions. They even included a Colab + Ngrok method to run Ollama in the cloud for free in one of the tutorials. The idea here is that an organization could run a model like DeepSeek-7B on an internal server to answer queries, avoiding external API calls altogether. DeepSeek’s performance (especially the 67B version) is quite strong in many tasks, meaning it can yield high-quality answers comparable to well-known models.

Integration for a local LLM would involve:

* Running the model in a server mode. For example, Ollama (or alternatives like HuggingFace’s text-generation-inference server) can host a model and provide a REST or socket API.
* Our backend would detect if LLM\_PROVIDER=deepseek or similar, and instead of calling a cloud API, it would call http://localhost:port/generate with a JSON including the prompt (or if using Ollama’s CLI, maybe spawn a subprocess – but a web service is easier).
* The local model might be slower (depending on hardware), so we might handle that by possibly reducing context or other ways. But the architecture remains the same logically.

**LLM Factory Implementation:**  
In code, we likely have something like:

class LLMFactory:

def \_\_init\_\_(self, provider:str):

self.provider = provider

if provider=="google":

self.client = GoogleLLMClient(api\_key=...)

elif provider=="deepseek":

self.client = LocalLLMClient(endpoint\_url=...)

# possibly an OpenAI client as another option if needed

def generate\_answer(self, prompt:str) -> str:

return self.client.generate(prompt)

This abstraction means elsewhere in the code, we don’t worry if it’s Google or DeepSeek – we just call llm\_factory.generate\_answer(prompt). The details are inside the client classes.

**Handling Response Differences:** Cloud APIs often return extra info (like usage tokens, etc.), while a local might just return text. We ensure to extract just the text. If the model’s output contains any special tokens or formatting (some models output markdown or cite sources if fine-tuned), we handle accordingly.

**Comparison and Rationale:**

* Using Google’s model could give very accurate answers but at the cost of sending data externally and incurring usage cost. It may also have rate limits and requires internet connectivity.
* Using DeepSeek (or any local open model) keeps data local and can be cost-effective after initial setup (no per-query cost, just infrastructure). The trade-off is one needs a capable machine (the 67B model likely needs a GPU server with a lot of memory). For a smaller setup (like on a personal PC), a 7B model could run but might not be as accurate.
* The project is likely demonstrating both to show that the system does not depend on proprietary models and can work with cutting-edge open-source ones.

It’s noteworthy that the mention of DeepSeek implies the documentation is up-to-date with late 2023/2024 developments in open LLMs. DeepSeek is a recent entrant (with papers in arXiv in 2023) and even mentioned in context of Ollama. We included it likely to highlight that our system can integrate “the newest open-source LLM with 67B parameters” which is a strong statement for a local RAG system’s potential (it means near state-of-the-art performance entirely locally, since DeepSeek 67B is reported to outperform Llama2 70B in some areas).

**Alternate LLMs:** The architecture is not locked to just these two; one could integrate OpenAI’s GPT easily (the tutorial probably did earlier on), or Microsoft Azure’s, etc. The concept of LLM Factory means new adapters can be added. So if tomorrow there’s a “BetterSeek LLM”, we could plug it in as long as we have a way to send prompt and get response.

**Prompting Strategy:** It’s worth mentioning again how we prompt the LLM affects the answer. We have to be careful to avoid the model ignoring the context or going off-track. Often, prefixing with an instruction helps. For example, we might include at the beginning of the prompt: “You are an expert assistant that answers questions based on provided documents. If the answer is not in the documents, say you don’t know. Do not fabricate information.” By doing this, we guide the LLM to use the context and be factual. In many RAG implementations, prompt engineering is iterative. In our project, we likely started with a straightforward approach as described earlier. Fine-tuning it could be future work (like trying chain-of-thought prompting or few-shot examples where we show the model an example of how to use context). Since this is a documentation focus, we outline that such an instruction was used.

**Example of Use:** Let’s walk through a user question to illustrate the integration:  
User asks: “What are catechins and how do they benefit health?” Suppose our context (from retrieval) has a snippet: “Catechins are a type of antioxidant found in green tea. Studies suggest they improve metabolism and reduce inflammation.” This is passed in prompt.

* If using Google’s LLM: the prompt goes to Google’s API. We get back something like: “Catechins are antioxidants. They benefit health by boosting metabolism and reducing inflammation, among other effects.” The backend returns that to user.
* If using DeepSeek: the prompt goes to local model. It might generate a slightly different phrasing but hopefully similar content, if the model is good. Since DeepSeek is open-source, if needed one could even fine-tune it on a Q&A style or the specific docs, but our approach is zero-shot (no fine-tuning, just prompting).
* In either case, the user receives an answer that directly references the content (implicitly or explicitly).

**Performance considerations:** Cloud LLM might have higher throughput but also latency due to network. Local LLM might be limited by hardware (67B might take several seconds per answer on a high-end GPU, 7B might be a bit faster but less accurate). We made sure the architecture can accommodate these differences. For instance, the FastAPI can handle async requests; if one query is waiting on the model, others could still be processed if on separate worker threads. We might configure the number of concurrent requests based on expectation.

In conclusion, the LLM integration portion of our system architecture ensures that we can plug in different answering engines without altering the rest of the pipeline. It provides the “brain” that synthesizes an answer from the retrieved facts. By accommodating both a cloud-based and a local model, we future-proof the system and respect various deployment needs (cloud vs on-premises). This design choice echoes a common theme in RAG applications: decoupling retrieval from generation so each can improve independently – one can update the knowledge base or switch models as needed without redesigning the whole system.

Having described all key components of the architecture – the pipeline, the database, the embedding service, and the LLM – the next chapter will move into a more concrete discussion of the actual implementation. We will examine the tools, frameworks, and libraries used (some already hinted, like FastAPI, SQLAlchemy, etc.) and how the codebase is structured across modules.

**5. Technology Stack and Tools**

In this chapter, we enumerate and describe each major tool, framework, and library used in the project. For each, we explain its role in the system, why it was chosen, and where in the codebase it is utilized. This provides a clear map of the technologies underpinning the Local RAG application.

**5.1 FastAPI (Backend Web Framework)**

**What it is:** FastAPI is a modern, high-performance web framework for building APIs with Python. It is known for its speed (built atop ASGI for async support) and developer-friendly features like automatic documentation, data validation via Pydantic, and dependency injection.

**Role in project:** FastAPI is the backbone of our backend server. It handles all HTTP requests from the frontend or clients. We defined various API endpoints using FastAPI decorators (like @app.post("/upload") for file uploads, @app.post("/query") for queries, etc.). FastAPI manages parsing incoming requests, calling our Python functions, and returning responses in JSON format (or serving files if needed). It also serves an auto-generated OpenAPI documentation (Swagger UI), which is useful for testing the API manually.

**Why chosen:** The project (mini-rag) specifically introduced FastAPI early on (in tutorial step 5), likely due to its ease of use and performance. FastAPI is a great fit for machine learning applications because it easily allows integration of async IO (useful if we want to do concurrent calls to external services like the Cohere API or the DB). Also, being relatively lightweight compared to something like Django, it’s suitable for a microservice-style application like ours which mainly just provides a JSON API. Its Pydantic model integration helps in validating input data (for instance, we could have a Pydantic model for a query request that ensures the query string is present and not too long, etc.).

**Where in codebase:** The main FastAPI app is likely instantiated in main.py within src/. For example, one might see:

app = FastAPI(title="Local RAG API", version="1.0")

We then include routers from different modules, e.g., app.include\_router(upload\_router) and app.include\_router(query\_router). These routers might be defined in src/routes/upload.py, src/routes/query.py etc., grouping endpoints logically. Each endpoint function uses type hints and Pydantic schemas for request/response data. For example:

@router.post("/query", response\_model=AnswerResponse)

async def query\_documents(request: QueryRequest):

# logic to handle query

...

Here, QueryRequest and AnswerResponse might be Pydantic models defined in src/models or src/schemas.

**Key FastAPI features utilized:**

* **Dependency Injection:** We might use FastAPI’s Depends to get database sessions (common pattern with SQLAlchemy). For instance, a function parameter like db: Session = Depends(get\_db) will automatically provide a database session (where get\_db is a dependency function that yields a session from our SessionLocal). This pattern was likely introduced around the time we switched to SQLAlchemy.
* **File uploads:** FastAPI makes it easy to accept file uploads using File and UploadFile. Our upload endpoint probably uses file: UploadFile = File(...) parameter. FastAPI handles the reading of file stream, and we can then process file.file or save it.
* **Background tasks:** Possibly we could use BackgroundTasks to handle heavy processing (like embedding a large file) without blocking the request. We might have chosen to do things synchronously for simplicity, but it’s an option.

**Location in code:** According to the repository listing, the src directory is likely where the FastAPI app and route definitions reside. We saw src/routes in the code listing, indicating route modules are present. For example, maybe src/routes/documents.py for upload, src/routes/qa.py for query-answer endpoints, etc. The structure might also include src/services for logic that the routes call (like a src/services/rag.py that orchestrates retrieval and generation given a query).

**FastAPI in development vs production:** For development, we run uvicorn main:app --reload ... as noted in README. In production (like Docker deployment), we might not use --reload and might set up Gunicorn or simply uvicorn without reload. The provided Docker likely sets up the service accordingly.

In summary, **FastAPI** is the core framework that allows our application logic to be exposed as web services, enabling integration with frontend or other clients. It is used extensively in the codebase for routing and request handling, making the backend both robust and interactive (with docs). The choice of FastAPI aligns with modern Python web development practices, especially suited for API-centric applications like ours.

**5.2 PostgreSQL Database and pgvector Extension**

**What it is:** PostgreSQL is a powerful open-source relational database management system. The **pgvector** extension is an add-on for Postgres that introduces a new column type for vectors and related operators to perform similarity search on those vectors. Essentially, pgvector allows Postgres to function as a vector database.

**Role in project:** PostgreSQL serves as the persistent storage for documents (and their embeddings) as well as any other data the system needs to store (for example, user info if multi-user, or system logs if we store them, etc.). Specifically, we use Postgres to store:

* Document metadata (filename, etc.) and content (or content chunks).
* Embedding vectors for each content chunk, using pgvector’s VECTOR type columns.
* Possibly Q&A history (if we chose to log questions and answers).
* The project also previously used MongoDB (in early steps) for storing files; by step 20, it moved to Postgres. Our branch (tut-014) might still have some Mongo references, but we assume by final integration we use Postgres. Indeed, step 20/21 of the tutorial (though beyond tut-014) introduces “From Mongo to Postgres + SQLAlchemy & Alembic” and “The way to PgVector”, implying the final system uses Postgres. Our documentation focuses on that final architecture.

**Why chosen:** PostgreSQL is known for reliability, ACID compliance, and rich features. Using Postgres with pgvector is an increasingly popular approach to implement RAG systems because it avoids needing a separate vector DB and leverages existing SQL ecosystem. For our project:

* We needed a way to do semantic vector search – pgvector provides that in Postgres, so it’s a natural fit.
* We also benefit from Postgres for other queries; for example, if we wanted to filter by document type or date or user, we can use regular SQL conditions combined with vector similarity in one query.
* Using a single data store simplifies deployment (just one database to maintain).
* The team behind mini-rag likely chose it to demonstrate how to integrate vector search into a traditional DB.

**Where in codebase:** The configuration for Postgres (connection details) likely lies in environment variables (like DATABASE\_URL or separate DB\_NAME, DB\_USER, DB\_PASS etc.). The .env.example in the repo presumably contains these (though we didn't see, the README shows commands copying .env files). The connection is set up via SQLAlchemy. Possibly in src/db.py or similar we have something like:

from sqlalchemy import create\_engine

from sqlalchemy.orm import sessionmaker

DATABASE\_URL = os.getenv("DATABASE\_URL")

engine = create\_engine(DATABASE\_URL)

SessionLocal = sessionmaker(bind=engine)

We also have models defined using SQLAlchemy’s ORM (or we could use Alembic to generate tables via migrations). The models might be in src/models.py or a package. For instance, a model for Document:

class Document(Base):

\_\_tablename\_\_ = 'documents'

id = Column(Integer, primary\_key=True)

title = Column(String)

...

chunks = relationship("DocumentChunk", back\_populates="document")

And a model for DocumentChunk:

class DocumentChunk(Base):

\_\_tablename\_\_ = 'document\_chunks'

id = Column(Integer, primary\_key=True)

document\_id = Column(ForeignKey('documents.id'))

content = Column(Text)

embedding = Column(Vector(768)) # from sqlalchemy-pgvector, perhaps

document = relationship("Document", back\_populates="chunks")

We then use these in code to add/read entries. For example, in an upload endpoint:

db\_doc = Document(title=file.filename, ...)

db.add(db\_doc); db.commit()

# Then for each chunk:

chunk = DocumentChunk(document\_id=db\_doc.id, content=chunk\_text, embedding=vector)

db.add(chunk)

db.commit()

For querying:

# Suppose we have query\_vector as a list or numpy array

results = db.query(DocumentChunk).order\_by(DocumentChunk.embedding.cosine\_distance(query\_vector)).limit(5).all()

If using the sqlalchemy-pgvector library, it might allow such query syntax (or we might have to use text for the <-> operator). In raw SQL, we could use session.execute("SELECT \* FROM document\_chunks ORDER BY embedding <-> :qvec LIMIT 5", params={"qvec": query\_vector}).

**pgvector usage:** We initialized the extension in the DB (maybe via Alembic migration: op.execute("CREATE EXTENSION IF NOT EXISTS vector")). We define the vector column. We create the index using migration as well:

op.create\_index('ix\_document\_chunks\_embedding\_vec', 'document\_chunks', [sa.text('embedding vector\_cosine\_ops')], postgresql\_using='ivfflat')

(This is SQLAlchemy Alembic style of creating an index with specific ops class and method.)

**Location of migrations:** The repository had an alembic directory presumably (not explicitly seen, but the README mentions Alembic migrations). Possibly there's a versions folder with migration scripts for creating the documents and chunks tables and adding pgvector.

**Database usage in development:** We likely used a local Postgres instance (for example, via Docker Compose as indicated by the docker directory containing presumably a docker-compose.yml with Postgres service). Indeed, the README instructs to cd docker && cp .env.example .env && docker compose up -d to run services, likely starting a Postgres container among others.

**pgvector alternatives considered:** If not pgvector, we could have used a standalone vector DB like Qdrant or Pinecone. The original tutorial did show Qdrant integration in step 15. Qdrant is a dedicated vector DB and might offer some performance benefits or easier vector-specific querying. However, integrating that means running an extra service and learning its API. By using pgvector, we kept the stack simpler. Also, Postgres was needed anyway to store relational info after moving off Mongo.

**Summary of PG & pgvector in code:**

* src/config.py or .env: contains DB connection settings.
* src/db.py: sets up engine and session and maybe Base (from declarative\_base()).
* src/models/ or similar: contains ORM classes with vector columns.
* src/routes or src/services: uses Session (via dependencies) to query/insert.
* docker/docker-compose.yml: defines a postgres service with an image like postgres:14 and environment to enable pgvector (some Postgres images come with pgvector preinstalled or we might have to run SQL to install it).
* requirements.txt: includes psycopg2 or asyncpg and sqlalchemy and sqlalchemy-pgvector.

**Important environment variables:** OPENAI\_API\_KEY was mentioned in the README for environment, but for Postgres, we likely have something like DATABASE\_URL=postgresql+psycopg2://user:pass@hostname/dbname. The .env.example in docker might contain POSTGRES\_USER, POSTGRES\_PASSWORD, POSTGRES\_DB which are used by the Postgres container and by our app to construct the URL.

In conclusion, **PostgreSQL with pgvector** is the cornerstone of our data layer. Its presence in the codebase is manifested through our ORM models and SQL queries that handle storing and retrieving document data. This technology choice provides both conventional database capabilities and the specialized semantic search feature that we need, all within one system, illustrating a pragmatic and powerful solution for our RAG application.

**5.3 Cohere Embeddings API**

**What it is:** The Cohere API is a cloud service providing NLP models, including text generation and text embedding. Specifically, the **Cohere Embeddings API** endpoint allows developers to send in text and receive high-dimensional vectors (embeddings) that capture the text’s semantic meaning. Cohere offers various embedding models (of differing sizes and capabilities) and requires an API key for access.

**Role in project:** Cohere’s embedding service is used to convert both document text and user queries into embedding vectors. This is a critical part of our system’s retrieval pipeline (as detailed in section 4.4). Whenever we ingest a document chunk or get a query, we call Cohere’s API to get its embedding. These embeddings are then stored or used for similarity search against stored embeddings. Essentially, Cohere’s service provides the “intelligence” to map language to vector space.

**Why chosen:**

* The project likely chose Cohere to diversify away from OpenAI and showcase another leading NLP platform. Cohere’s embeddings are known to be high quality, and their terms might be more flexible.
* Also, the mention of Cohere aligns with the narrative of building a self-contained RAG system perhaps with non-OpenAI dependency.
* Technical reason: the Cohere API is straightforward and the free tier (if available) might allow some experimentation. For academic projects, Cohere might provide credits.
* It's also possible the original mini-rag started with OpenAI embeddings (given the .env mentioned OPENAI\_API\_KEY), and at some point (perhaps when introducing local LLM) they switched to demonstrate another service (Cohere). Regardless, our documentation focuses on Cohere as per user request.

**Where in codebase:** We likely have a module like src/services/embedding.py or something similar, where a function get\_embedding(text: str) -> list[float] is defined. Implementation might use cohere.Client. If we listed requirements, likely cohere SDK is included. The code would look something like:

import cohere

co = cohere.Client(os.getenv("COHERE\_API\_KEY"))

def get\_embedding(text: str) -> List[float]:

response = co.embed(texts=[text])

return response.embeddings[0]

We might also have a batch version:

def get\_embeddings(texts: List[str]) -> List[List[float]]:

response = co.embed(texts=texts)

return response.embeddings

These functions are called in the document ingestion flow (for each chunk, or perhaps batch chunks 5 or 10 at a time to be efficient) and in query handling (single text).

If we didn’t use the official SDK, we might do a direct HTTP call using requests:

import requests

COHERE\_API\_KEY = os.getenv("COHERE\_API\_KEY")

headers = {"Authorization": f"Bearer {COHERE\_API\_KEY}"}

data = {"texts": [text]}

res = requests.post("https://api.cohere.ai/v1/embed", json=data, headers=headers)

embedding = res.json()["embeddings"][0]

But the SDK is convenient.

**Handling errors:** The code should handle exceptions from the Cohere API. If the API key is invalid or if the service is unreachable, we might log an error and propagate an HTTP 500 error to the user query. In a production scenario, we might want to catch that and maybe try an alternate (if configured), but in our context we likely assume the key is correct.

**Cohere Model choice:** We likely specify which Cohere embed model to use when calling. The default might be their medium model. Possibly we allow configuration via env (like COHERE\_MODEL=embed-english-v2.0). If not specified, the API might pick a default. It’s better to be explicit. In code:

co.embed(texts=[text], model="embed-english-v2.0", truncate="END")

(for example, Cohere allows specifying a truncate mode if text is longer than model max).

**Performance:** The embedding call latency depends on text length and model. Usually a single embedding call is on the order of 100ms for short text (plus network overhead). Batch calls might be a bit more but better than multiple sequential calls. Our system might embed maybe up to a few hundred chunks on a big document upload (which could take a number of seconds). For queries, 0.2-0.5s to embed the question is fine.

**Cost:** We should note that using Cohere’s API is a paid service beyond a free tier. Each API call costs some credits (embedding is cheaper than generation usually). The project might have used a trial or has minimal usage. But in documentation, one might mention that this could incur cost and to monitor usage.

**Where configured:** The API key is likely in the .env file as COHERE\_API\_KEY. It should be kept private. In docker/.env.example they might have placeholders. When running locally, the user has to set their key. We also might have logic to ensure it’s present on startup (FastAPI could check or during first embed call, throw error if not set).

**Alternatives:** If not Cohere, OpenAI’s embedding model could be used similarly (with openai library and openai.Embedding.create). The code structure would be similar. We could have even abstracted it: e.g. an EmbeddingFactory to allow switching provider. The question specifically includes Cohere, so we focus on that.

**Associated sources in documentation:** We cited Cohere docs lines which define embeddings and usage in semantic search context; this frames why we are using it. That line is likely directly relevant to how we use them.

In summary, the **Cohere Embeddings API** is an external dependency integral to our semantic search functionality. In the code, it appears wherever we translate text to vectors (document ingestion and query processing). Its presence is signaled by the use of the Cohere client or HTTP calls, and configured by an API key in environment variables. This tool exemplifies how our system leverages state-of-the-art NLP-as-a-service to achieve intelligent behavior without having to develop our own machine learning models from scratch.

**5.4 Google PaLM API / DeepSeek LLM (Answer Generation)**

*(We break this into two parts since they are distinct tools, but they share a similar role.)*

**Google PaLM API**:  
**What it is:** Google’s PaLM is a large language model developed by Google, accessible via their Cloud services. The PaLM API (or Vertex AI Generative service) allows developers to get text completions from Google’s models. Google’s models are comparable to OpenAI’s in capability. Using them typically requires GCP credentials and adherence to certain usage policies.

**Role in project:** In our project, Google’s PaLM API is one option for the LLM that generates answers in the RAG pipeline. When configured to use Google, our system will send the composed question+context prompt to Google’s API and retrieve the model’s answer. The answer is then returned to the user. This provides a powerful cloud-based brain for the system, likely yielding high-quality responses.

**Why chosen:** Including Google’s LLM serves several purposes:

* It gives an alternative to OpenAI, showing versatility.
* Google’s model might be preferred for integration if a user is already on GCP or if they find it more reliable or cost-effective.
* It demonstrates how one can integrate a third-party LLM via API in general, a pattern which could be extended to others.

**Where in codebase:** There might be a src/services/llm\_google.py or something similar implementing a client for Google. Possibly using Google’s official SDK (like the google.cloud AI library) or raw HTTP. For example, using google.generativeai library which Google released:

import google.generativeai as palm

palm.configure(api\_key=API\_KEY)

response = palm.generate\_text(prompt=prompt, model="models/text-bison-001", temperature=0.7)

answer = response.result

Alternatively, if using Vertex AI:

from google.cloud import aiplatform

answer = aiplatform.GenerativeAiService.predict(... prompt=prompt ...)

The specific implementation depends on how Google exposes it (the PaLM API I believe has a REST endpoint as well).  
We likely have an environment variable GOOGLE\_API\_KEY or a service account JSON file for auth.

In code, if we didn't use an SDK, we might do:

requests.post("https://generativelanguage.googleapis.com/v1beta2/models/text-bison-001:generateText?key=API\_KEY",

json={"prompt": {"text": prompt}, "temperature":0.7, "candidateCount":1})

This would return a JSON containing candidates.

**Integration point:** The integration is triggered in the LLM Factory or a conditional in the query pipeline. Pseudocode:

if LLM\_PROVIDER == "google":

answer\_text = google\_client.generate(prompt)

elif LLM\_PROVIDER == "deepseek":

answer\_text = deepseek\_client.generate(prompt)

We ensure both yield a plain string answer.

**DeepSeek LLM (Local)**:  
**What it is:** DeepSeek LLM refers to an open-source large language model (they have 7B and 67B versions as per their GitHub). It can be run locally given enough compute. The project references it likely as an example of a local LLM that can be used instead of cloud APIs.

**Role in project:** When configured for DeepSeek, the system will route the prompt to a local inference server running DeepSeek. We possibly use **Ollama** or similar as mentioned. Ollama is a CLI for running models like Llama, and they have presumably integrated DeepSeek models (since the search results mentioned DeepSeek on Ollama). So our backend might open a subprocess to call ollama generate "deepseek:7b" -p "{prompt}", or if Ollama has a REST endpoint, use that.

Alternatively, we might use the HuggingFace transformers pipeline directly in Python to run a smaller model (though 67B is too large to run without specialized setup). However, the tutorial likely used the trick of running Ollama in Colab with Ngrok to simulate a local LLM server for demonstration.

**Why chosen:** Incorporating DeepSeek demonstrates the ability to keep the entire pipeline local (no external calls for generation). It also rides on the trend of powerful open models becoming available, meaning one can have a private ChatGPT-like system. For a graduation project, this is a compelling angle: showing that with appropriate hardware, one could avoid any API and still get great results. It underscores data privacy and control.

**Where in codebase:** Possibly a src/services/llm\_local.py implementing local LLM calls. This could be as simple as:

def generate\_with\_ollama(prompt: str) -> str:

response = requests.post("http://localhost:11434/generate", json={"model": "DeepSeek-67B", "prompt": prompt})

return response.json()["generated\_text"]

This is hypothetical; if using Ollama’s API (Ollama listens on localhost:11434 by default). If not, maybe they call the binary:

result = subprocess.run(["ollama", "run", "deepseek", "--prompt", prompt], capture\_output=True)

text = result.stdout

However, calling subprocess for each query might be slow; an API or persistent server is better.

We should also consider, if not using Ollama, maybe using a HuggingFace pipeline:

from transformers import AutoModelForCausalLM, AutoTokenizer

model = AutoModelForCausalLM.from\_pretrained("deepseek/deepseek-llm-7b")

tokenizer = AutoTokenizer.from\_pretrained("deepseek/deepseek-llm-7b")

inputs = tokenizer(prompt, return\_tensors='pt')

outputs = model.generate(\*\*inputs, max\_new\_tokens=200)

text = tokenizer.decode(outputs[0], skip\_special\_tokens=True)

But doing that synchronously in the API for each call could be heavy, unless we keep the model loaded globally (which we can, at startup load the model into memory, which for 7B might be borderline in CPU but possible with enough RAM, for 67B definitely requires a GPU and lots of RAM).

Given complexity, likely the recommended approach was to rely on an external tool like Ollama and just call it.

**Configuration:** There might be environment flags such as USE\_LOCAL\_LLM=True or LLM\_PROVIDER=deepseek to switch. The code probably in config picks one based on presence of keys:  
For instance, if GOOGLE\_API\_KEY provided, use Google, elif OPENAI\_API\_KEY use OpenAI, elif none but a local model is present, use local.

**Utilization in practice:** When using local, one might restrict usage to smaller queries or admonish that it's slower. Possibly, the project expects the user to manually start the local LLM server (like they mention in README optional steps: *"(Optional) Run Ollama Local LLM Server using Colab + Ngrok"*).

**Differences in output handling:** The local model may output just raw text. The Google API might return a structure. Our code normalizes that to just an answer string.

**Testing:** We likely tested the pipeline with the local mode in a limited fashion (maybe just see it works for a simple query). It's possible that the default mode remains using an API (since running DeepSeek 67B is non-trivial outside their demonstration environment).

**Conclusion of LLMs:** The combination of Google PaLM and DeepSeek demonstrates hybrid capability:

* Cloud Option (Google PaLM): for maximum performance with dependency on external service.
* Local Option (DeepSeek): for maximum privacy and offline capability, at the cost of needing heavy resources.

In code, these are abstracted as pluggable backends. The existence of these in the code is seen through mentions of either Google’s library or http calls to local server. Also environment variables or config classes for each.

By offering both, our project is robust and can cater to different deployment scenarios:

* If internet access and API budget are available, use Google (or one could substitute OpenAI similarly).
* If not, and one has a strong machine, use an open model like DeepSeek.

This design is an important demonstration of the **Retrieval-Augmented Generation** concept: the retrieval part remains same, and generation can be done by any capable LLM. Thus, our system is not tied to a single AI provider.

**5.5 Frontend Framework (Web Interface)**

**What it is:** The project includes a **frontend** component (as indicated by the frontend directory in the repository). Based on typical stacks and the presence of package-lock.json, it’s likely a JavaScript/TypeScript single-page application. It might be using a framework like React, or possibly Svelte or Vue. Given the timeline of the tutorial and the community, my guess is it could be a simple React app (or even just HTML + some JS fetch calls).

**Role in project:** The frontend provides a user-friendly interface to interact with the system. It typically allows:

* Uploading documents (maybe via a form or drag-and-drop).
* Listing the uploaded documents or their status.
* Entering questions in a chat-like or form interface.
* Displaying the answers returned by the backend, possibly along with source citations.

In essence, it turns our backend API into an application that a non-technical user can use in a browser.

**Why chosen:** While one could use the API via cURL or Postman, a frontend makes the demonstration of the project much more compelling (especially for a graduation project, showing a polished UI). It also allows iterative Q&A like a chat, which is the natural way to use such a system. The project likely included it to complete the end-to-end user experience.

**Where in codebase:** The frontend directory contains the frontend code. If it's a React app (just an assumption), inside might be src/ with components, or if it's Next.js it might have pages. The package-lock implies it uses npm. The README says cd frontend && npm run dev to start it for development. So it’s likely a Vite or Create-React-App or Next dev server.

Potential clues: The tutorial being in Arabic, maybe they built a simple UI in React. The actual content of that directory is not shown in our browse output, but it's safe to assume:

* There is an interface for uploading files: either a file input or using an API endpoint (the backend likely has an upload route).
* Possibly a simple list of uploaded docs (maybe just to confirm upload success).
* A text area or input for queries and a submit button.
* An area to display the answer. Possibly including some formatting (maybe bolding parts, or listing sources).
* They might also show the context or highlight which document was used, not sure but could be.

**Tech stack guess:** If it's React, they might have used some UI library for layout. If it's Svelte (less likely but possible given Svelte’s popularity for small apps), then it would have a different structure.  
Given the mention of “npm run dev”, which is common to many frameworks, I'll lean on React + perhaps Vite as bundler (since Vite is common and fast).

**Communication with backend:** The frontend likely uses the browser Fetch API or axios to call the FastAPI endpoints. For example, on file upload:

const formData = new FormData();

formData.append('file', selectedFile);

fetch('/upload', {method: 'POST', body: formData});

Since backend is at localhost:5000 (as uvicorn default), and frontend dev runs at perhaps localhost:3000, we might have to deal with CORS. Possibly the FastAPI app has from fastapi.middleware.cors import CORSMiddleware and allows the frontend origin.

The docker-compose might serve both, or the production build of frontend might be served by an Nginx container or via FastAPI (embedding static files). But given separation, maybe not.

**Where to run:** In development, you run backend (on port 5000) and npm run dev for front (on 5173 or 3000 depending on environment). In production, likely they built the frontend (npm run build) and copied the static files to a web server container or served via a static route in FastAPI.

However, the docker-compose in docker might be configured to spin up something for the frontend too. Possibly using a Node container to serve it or a static volume with Nginx. It's speculative, but we might find in docker/docker-compose.yml some service like frontend with a Dockerfile pointing to building the React app.

**User experience**:

* The user first sees an upload button. They add some documents. The UI might show a success or list the file names in a section "Documents".
* Then a text input asks "Ask a question about your documents". They type a question and hit enter.
* The UI disables input, shows a "Loading..." or spinner while awaiting response.
* After getting answer, it displays the answer text. If sources are provided, maybe shows them as footnotes or links.
* Possibly, since it’s akin to a chatbot, they might show each Q&A exchange in a chat bubble format, allowing multiple questions sequentially. The requirement didn't explicitly mention multi-turn dialogues, but a user can always ask a follow-up question provided the previous Q doesn't persist state (our system doesn't inherently do conversational memory, although one could feed the last answer as part of context if desired).

**Styling and polish:** They may have kept it simple with basic bootstrap or CSS. If it’s a hacky UI, that might be fine for demonstration.

**Why front-end tech selection matters:** If using React or Vue, it shows familiarity with modern frameworks. If just pure HTML, it’s simpler but less interactive (though one can do a lot with plain JS). The presence of package-lock suggests use of npm packages, thus likely a framework.

**Notable library:** Possibly they used **Swagger UI** for file upload. But since they specifically instruct npm run dev, it’s a custom front-end, not just relying on Swagger. Swagger (FastAPI docs) can upload files too for testing, but not user-friendly for non-devs.

**Integration with back-end in deployment:** The base URL for API might need to be configured (in dev you might proxy from front to backend to avoid CORS). In build, maybe the front expects the API at same domain, maybe they serve it under e.g. path /api. Some front-end frameworks allow a proxy in dev (like CRA allows a "proxy": "http://localhost:5000" in package.json).  
Perhaps the project didn't get too deep into deployment nuance since it’s educational.

**Conclusion on Frontend:** It is a crucial part that turns our system from code to an application. In codebase, it is separated and built with Node. The documentation might not detail the frontend code much, but for completeness of technical documentation, we describe its presence and general function.

Its inclusion is also evident from instructions in README about running it and in the docker/ environment.

**5.6 Docker and Deployment Configuration**

**What it is:** Docker is a containerization platform. The project includes a docker directory which suggests they prepared Docker configurations for running services (likely the database, possibly the backend, etc.). Docker Compose might be used to tie them together. This is typical to ensure the project can run easily on any environment without manual setup of DB, etc.

**Role in project:** Docker is used to encapsulate the environment for the application. Likely:

* A **Dockerfile** for the backend (FastAPI app) specifying base image (python:3.10 maybe), copying source code, installing dependencies (pip install -r requirements.txt), and setting up to run uvicorn main:app on container start.
* Possibly a **Dockerfile** or direct image use for the database (Postgres). Usually, we just reference the official postgres image in docker-compose.
* Possibly a service for the front-end. If they went with multi-container deployment, maybe they use Nginx or something to serve the static built frontend or run the dev server. Alternatively, they might not containerize the dev front-end (since one would normally just build for prod and serve static).
* Also, maybe a vector DB container if they had gone with Qdrant earlier, but now replaced by Postgres/pgvector, so likely not needed.

The presence of docker-compose.yml is highly likely given the README instructions. The README snippet:

$ cd docker

$ cp .env.example .env

# update .env with credentials

$ sudo docker compose up -d

This suggests in docker/.env.example there are env values used by docker-compose (like for Postgres environment and maybe for app environment variables).

**Why chosen:** Docker ensures that the development and production environment are consistent. It's easier for others to run (just install Docker and docker-compose, then up). It's also beneficial for deployment to cloud (maybe they deploy to a VM or something by running these containers). It encapsulates dependencies (like correct Python version, all pip packages, pgvector extension, etc.).

**Where in codebase:** The docker folder presumably contains:

* docker-compose.yml
* .env.example
* Possibly Dockerfile for app.
* Maybe Dockerfile.frontend if needed.
* Possibly scripts.

We see in the repo listing:

67 .vscode/

68 docker/

69 frontend/

So, docker is at same level as code, likely containing those config files.

**Docker Compose specifics (inferred):**  
Likely services:

services:

db:

image: postgres:15

environment:

- POSTGRES\_USER=${POSTGRES\_USER}

- POSTGRES\_PASSWORD=${POSTGRES\_PASSWORD}

- POSTGRES\_DB=${POSTGRES\_DB}

volumes:

- db\_data:/var/lib/postgresql/data

ports:

- "5432:5432"

backend:

build: ../ (build context likely root where Dockerfile is)

ports:

- "5000:5000"

env\_file:

- ../.env (with keys like COHERE\_API\_KEY, DB credentials if needed by app)

depends\_on:

- db

frontend:

build: ../frontend (if they containerize the UI)

ports:

- "3000:3000"

depends\_on:

- backend

This is hypothetical. However, often one might not containerize the dev UI, instead just run npm run build and serve static. If time, they might have integrated a production server for static files.

Alternatively, an easier approach:

* After building front, copy frontend/dist into a folder that an Nginx container serves, or even simpler: have the FastAPI app serve static files (Starlette allows mounting StaticFiles). Actually, if we check frontend commands, there's no explicit building in README aside from dev run. Possibly they expected to run dev for demonstration (less likely for final though).
* The project might not emphasize full production readiness of front; the focus is more on the RAG pipeline. But because they mention 100-page documentation, likely they did put some emphasis on packaging everything.

**.env files in docker:** Typically:  
docker/.env.example might contain:

POSTGRES\_USER=postgres

POSTGRES\_PASSWORD=postgres

POSTGRES\_DB=mini\_rag

OPENAI\_API\_KEY=sk-...

COHERE\_API\_KEY=...

GOOGLE\_API\_KEY=...

and so on, providing a template for the user.

**Alembic in Docker:** They might handle running migrations in container entrypoint. Possibly the backend container’s entrypoint script runs alembic upgrade head (since README instructs to do it manually too, maybe in container scenario they automated it or still require manual).

**Edge: Embedding services** – If using Cohere, our backend container calls out to Cohere API (no special config needed except key). If using local DeepSeek, presumably one must run an Ollama server outside of compose (or maybe they could run an Ollama container if available, but likely out of scope). They do mention using Colab+Ngrok for local LLM, which is separate from Compose.

**Continuous integration** – not mentioned, but containerization often suggests maybe a path to CI or ease of testing. Possibly not set up in code explicitly, but this approach is modern best practice.

**Volume usage** – for database persistence (so that container restart doesn’t wipe DB). Usually define a volume for Postgres data.

**Conclusion on Docker:** Docker and docker-compose are used to streamline the deployment of all components. In code, references to environment variables (like in config.py or settings) correspond to those from .env which docker-compose passes in. The presence of Docker config indicates that the project can be run reproducibly by others, which is important in an academic or demo setting.

By summarizing the tools and frameworks in this section, we have highlighted:

* The frameworks (FastAPI, front-end library) that structure the application logic and interface.
* The infrastructure components (Postgres with pgvector, Docker) that enable the system’s functionality and distribution.
* The external AI services (Cohere, Google PaLM) and open-source AI models (DeepSeek) that provide the machine learning intelligence.

Each tool was chosen to fulfill a specific requirement, and together they form a cohesive stack for building a modern RAG system. In the next chapter, we will explore the code organization in more depth, walking through how the modules and folders correspond to these tools and to the application features.

**6. Codebase Structure and Module Walkthrough**

This chapter provides an in-depth walkthrough of the project’s code organization. We will go through each major folder and module, explaining its purpose and how it fits into the overall system. By mapping out the codebase, one can better understand how the concepts discussed earlier are implemented in practice.

**6.1 Directory Layout Overview**

The repository is organized in a logical manner, separating concerns into different folders. Below is an approximate structure (simplified for clarity):

local\_rag/

├── src/

│ ├── main.py

│ ├── config.py (or settings.py)

│ ├── db.py (database setup)

│ ├── models/ (or a models.py)

│ │ ├── \_\_init\_\_.py

│ │ ├── document.py

│ │ ├── chunk.py

│ │ └── ... (possibly other models or Pydantic schemas)

│ ├── routes/ (or controllers/)

│ │ ├── \_\_init\_\_.py

│ │ ├── documents.py (for upload endpoints)

│ │ ├── query.py (for query endpoints)

│ │ └── ... (maybe a healthcheck or auth if any)

│ ├── services/ (business logic)

│ │ ├── embedding\_service.py

│ │ ├── rag\_service.py (or qa\_service.py)

│ │ ├── llm\_factory.py

│ │ ├── llm\_google.py

│ │ ├── llm\_local.py

│ │ └── ... (maybe file processing utilities)

│ ├── utils/ (if any utility functions)

│ └── ...

├── frontend/

│ ├── src/ (React/Vue code)

│ └── public/ (static assets, etc.)

├── docker/

│ ├── Dockerfile (for backend)

│ ├── docker-compose.yml

│ ├── .env.example

│ └── ...

├── alembic/ (if migrations are included)

│ ├── env.py

│ ├── versions/

│ │ └── xxxx\_create\_tables.py

│ └── ...

├── requirements.txt

├── README.md

└── ... (other root files like LICENSE, PRD.txt)

Now, let's go through the important parts of this structure.

**6.2 Models and Schemas (Data Layer)**

**Location:** Likely in src/models or directly in src/ as a models.py. These define the shape of data in both the database (SQLAlchemy models) and possibly the API layer (Pydantic models for request/response bodies).

**Purpose:** Models define how documents and other entities are stored, and schemas define how data is exchanged via the API.

* **Database Models (ORM):**
  + Document model representing an uploaded document. Fields: id, title/filename, maybe an original content field (or just link to chunks), possibly metadata like upload\_date.
  + DocumentChunk model representing a chunk of a document with fields: id, document\_id (FK), content (text of chunk), embedding (vector). Possibly also an index or position field.
  + If we had user accounts (unlikely given scope), there could be a User model.
  + If tracking Q&A interactions, maybe a Question model referencing a document or simply logging queries – but none was mentioned, likely omitted.

These models use SQLAlchemy Base = declarative\_base() and define \_\_tablename\_\_, columns (with pgvector for embedding as mentioned). They also can define relationships between Document and DocumentChunk (one-to-many).

* **Pydantic Models (Schemas):**
  + For requests: e.g. QueryRequest with field question: str.
  + For responses: AnswerResponse with fields like answer: str and maybe sources: List[Source].
  + Possibly DocumentUploadResponse which might just confirm success or return document id.
  + If they want to show which documents were used, they could have a Source schema with fields like document\_title and chunk\_snippet.
  + If multiple questions support, maybe QAHistory but probably not needed.

These are likely defined either in a separate schemas.py or within route modules for convenience. Pydantic ensures data validation (e.g. question not empty) and automatic docs in OpenAPI.

**Where referenced:**

* ORM models are used in the database operations (in services or directly in routes) – for instance, when saving chunks, we instantiate a DocumentChunk.
* Pydantic models are used as type hints in FastAPI routes and to shape output. For example:
* @router.post("/query", response\_model=AnswerResponse)
* def query\_documents(request: QueryRequest, db: Session = Depends(get\_db)):
* ...
* return AnswerResponse(answer=answer\_text, sources=source\_list)

FastAPI will convert that to JSON and validate types.

**Significance:**  
Defining these clearly is crucial because the entire pipeline’s data integrity rests on them. They ensure that:

* All embeddings are associated with a document and can be retrieved.
* All responses conform to a format (for frontend consumption or for users).
* They also serve as documentation for what data our API expects and returns.

**Note on Embedding data type:** Using Vector from pgvector extension in ORM (if using sqlalchemy-pgvector) is part of model definition for DocumentChunk. We might have:

from pgvector.sqlalchemy import Vector

embedding = Column(Vector(dim), nullable=False)

Where dim is numeric (e.g., 768). The code might have dim = 768 defined globally or obtained from config.

In summary, the **models** module captures our domain – documents and chunks – and the **schemas** capture the API interface for questions and answers. They form the foundation of how data is stored and communicated.

**6.3 API Routes and Controllers**

**Location:** src/routes directory containing modules like documents.py and query.py. Alternatively, some call this controllers or just put in main, but given we saw multiple branch references to “Nested Routes” in the tutorial, they likely organized routes in modules with APIRouters.

**Purpose:** These modules define the HTTP endpoints of the application. They use FastAPI’s APIRouter to group related endpoints. For example:

* documents.py might handle uploading and listing documents.
* query.py handles question answering endpoint(s).
* Possibly a health.py for a health-check (common in docker setups).
* If they had user management (not indicated), would be another router.

**Contents and Implementation:**

* **Document Upload Endpoint:**
* router = APIRouter(prefix="/documents", tags=["documents"])
* @router.post("/upload", response\_model=UploadResponse)
* async def upload\_document(file: UploadFile = File(...), db: Session = Depends(get\_db)):
* # 1. Save file content to text (if needed, or process stream)
* content\_bytes = await file.read()
* text = convert\_to\_text(file.filename, content\_bytes) # hypothetical util to extract text from PDF/Docx
* # 2. Chunk text
* chunks = split\_text\_to\_chunks(text)
* # 3. Create Document record
* doc = Document(title=file.filename)
* db.add(doc); db.commit(); db.refresh(doc)
* # 4. Embed chunks and create DocumentChunk records
* for chunk\_text in chunks:
* vector = get\_embedding(chunk\_text)
* chunk\_rec = DocumentChunk(document\_id=doc.id, content=chunk\_text, embedding=vector)
* db.add(chunk\_rec)
* db.commit()
* return UploadResponse(document\_id=doc.id, chunk\_count=len(chunks))

This example flow covers reading the file, parsing to text, chunking, embedding (calls service function), and storing. In reality, they might have offloaded some of this logic to a service function (for cleanliness and testability). For instance, rag\_service.py might have a function ingest\_document(file). But it's shown inline for clarity.

Also note: reading the entire file might not be ideal for large files (we could stream and chunk incrementally), but given simplicity, it's fine.

For converting PDF to text, perhaps they used a library like PyMuPDF or pdfminer, but the project may have limited scope. The tutorial might have shown using PyMuPDF or similar (depending on target audience). If no special library, maybe they restricted to plain text files for simplicity. However, the video list mentions "File Processing" at step 8 and "Mongo - Motor - File Upload" around step 9, implying some processing was done on various file types.

There's also mention of "Nested Routes + Env Values" at step 6, likely covering how to nest routers and how to use .env (for keys) – which lines up with our environment usage for keys.

* **Query/Answer Endpoint:**
* router = APIRouter(prefix="/query", tags=["query"])
* @router.post("/", response\_model=AnswerResponse)
* def answer\_query(request: QueryRequest, db: Session = Depends(get\_db)):
* question = request.question
* # 1. Embed the question
* q\_vector = get\_embedding(question)
* # 2. Retrieve similar chunks
* results = db.query(DocumentChunk)\
* .order\_by(DocumentChunk.embedding.cosine\_distance(q\_vector))\
* .limit(5).all()
* # 3. Prepare prompt from results
* context\_snippets = [res.content for res in results]
* prompt = build\_prompt(question, context\_snippets)
* # 4. Generate answer via LLM
* answer\_text = llm\_factory.generate\_answer(prompt)
* # 5. Optionally, prepare source info
* sources = []
* for res in results:
* sources.append(Source(document\_title=res.document.title, snippet=res.content[:100]))
* return AnswerResponse(answer=answer\_text, sources=sources)

Again, they might have moved some of these steps to a service function for cleanliness (e.g., rag\_service.answer\_question(question) which does 1-4).

Breaking it down:

* + Step 1: embed query (calls embedding\_service, likely synchronous since using sync DB operations).
  + Step 2: DB query for nearest chunks. We use cosine\_distance because we probably created index with vector\_cosine\_ops. If using raw text in query, they might have done something like db.execute("SELECT ... ORDER BY embedding <-> %s", [q\_vector]). But the pgvector SQLAlchemy integration might provide a function or comparator like shown.
  + Step 3: Build the prompt with context. The build\_prompt likely formats the question and each snippet with identifiers as described earlier. Possibly they label them [1], [2] etc. or just put them all under "Context: ...".
  + Step 4: Use our LLM Factory (which inside uses either Google or local or OpenAI depending config).
  + Step 5: Prepare sources list for output. They decide what to include: could be doc title and maybe a small excerpt. If not needed, they could simply output answer and maybe the count or nothing else. The user’s requirement mentions "with reference for all cited sources", so they likely included sources in answer or separately. Perhaps in AnswerResponse they have a list of document titles or IDs. Or they might have actually appended citations in the answer text (like "Green tea boosts metabolism [Document: HealthBenefits.pdf]"). But a cleaner way is separate field.  
    We'll assume they went with separate sources list.
  + Finally returns the AnswerResponse which FastAPI turns to JSON. For example:
  + {
  + "answer": "Green tea contains antioxidants called catechins that can boost metabolism and reduce disease risk.",
  + "sources": [
  + {"document\_title": "health.pdf", "snippet": "Green tea contains catechins..."},
  + {"document\_title": "nutrition.txt", "snippet": "A catechin is a natural antioxidant..."}
  + ]
  + }

**Dependency injection and config in routes:**  
We see db: Session = Depends(get\_db) in both endpoints to get a database session from SessionLocal. get\_db is often defined in db.py:

def get\_db():

db = SessionLocal()

try:

yield db

finally:

db.close()

This is the FastAPI way to ensure we get a fresh session and close it after request.

They might also use dependency for LLM Factory if desired, but likely it's a global object they reference (maybe created in main or in a service module).

**Integration between routes and services:**  
Some teams prefer to keep route functions thin and call service layer functions. Given the educational nature, they might have written more inline code in routes for demonstration, or they might have abstracted after initial demonstration.  
Given branch 14 was about LLM Factory, they were adding abstractions.

Perhaps:

* services/rag\_service.py with ingest\_document(file, db) and answer\_question(question, db) to encapsulate logic.
* Then routes/documents.py just calls rag\_service.ingest\_document(file, db) and returns result.
* routes/query.py calls rag\_service.answer\_question(question, db).

This separation makes testing easier (service functions can be tested without HTTP context) and keeps controllers clean.

**Routes summary:**

* Document upload route(s): Maybe one for single file upload, possibly one to list documents or fetch a document (if they built any UI for listing or if needed by front-end). They might have a GET /documents to return list of docs (with their titles and maybe count of chunks or something) so front-end could display what's uploaded. Or front-end may just not bother listing, and assume immediate usage after upload.
* Query route: The main Q&A endpoint. Perhaps they also allowed GET method for query via query param (less likely; usually a POST is used to allow long questions in body).
* Possibly a route to delete documents if needed (if UI had it). But not mentioned, so maybe not.
* Health route: Some projects include a simple @app.get("/health") that returns 200 to indicate service is up (useful for orchestration or readiness checks). It’s trivial to add, so maybe they did.

**Mounted routers in main.py:**  
In main.py, after creating app = FastAPI(), they likely have:

from routes import documents, query

app.include\_router(documents.router)

app.include\_router(query.router)

This mounts them so their prefixes work.

They also possibly add CORS middleware in main for frontend:

from fastapi.middleware.cors import CORSMiddleware

app.add\_middleware(

CORSMiddleware,

allow\_origins=["\*"], # in dev maybe open, in prod specify

allow\_methods=["\*"],

allow\_headers=["\*"]

)

That might have been set up in an earlier step when hooking up front-end.

**Conclusion on routes:** The route modules are essentially the interface of the backend. They coordinate between receiving HTTP input (files or JSON), invoking the appropriate processing (embedding, DB access, LLM calls), and returning the output in structured form. They handle any request-specific logic, such as reading file streams or merging outputs into final response. Their implementation confirms how the theoretical pipeline is executed in code.

**6.4 Service Layer (Processing and Utilities)**

**Location:** src/services or similar. This layer includes modules that implement the core logic without being tied to HTTP. They are invoked by the routes.

**Purpose:** Encapsulate complex operations like document parsing, text splitting, embedding calls, and LLM queries. This separation makes the code more modular and testable.

Key service components likely include:

* **File/Text Processing Utility:** Perhaps a function to convert various file types to raw text (convert\_to\_text(filename, bytes) -> str). If the project supports PDFs, this uses a PDF parser (maybe PyMuPDF fitz library). For .docx, perhaps python-docx or fall back to plain text. For .txt just decode.

If such a parser is used, it might be in a utils file, since it’s not core RAG but supporting.

Also, **split\_text\_to\_chunks(text: str) -> List[str]** might be in a utils or in rag\_service. Implementation might break text by paragraphs or around 200-300 words:  
For example:

import re

def split\_text\_to\_chunks(text, max\_len=500):

# split by double newlines or periods ensuring not to exceed max\_len

paragraphs = text.split("\n\n")

chunks = []

for para in paragraphs:

if len(para) <= max\_len:

chunks.append(para)

else:

# further split by sentence if needed

sentences = re.split(r'(?<=[.!?]) +', para)

chunk = ""

for sent in sentences:

if len(chunk) + len(sent) < max\_len:

chunk += sent + " "

else:

chunks.append(chunk.strip())

chunk = sent + " "

if chunk:

chunks.append(chunk.strip())

return chunks

This is just an example of how one might chunk. The actual algorithm might differ (some use fixed-size sliding window, etc.). But likely they did something simple like above.

* **Embedding Service:** (embedding\_service.py) – Contains the code to call Cohere’s API (or OpenAI if alternate).  
  Possibly a class EmbeddingService with a method embed(texts: List[str]). Or just standalone functions get\_embedding(text) and get\_embeddings(texts).  
  It will use the Cohere API key from config (so likely imports config.py where COHERE\_API\_KEY is stored).  
  It might also handle batching logic: e.g., if more than X texts, split into multiple API calls to not hit length limit or rate limit.

Could also include a fallback: if one embedding provider fails, try another (less likely needed).  
Might also normalizes the output (ensuring it's list of floats, maybe rounding or converting to list from numpy if needed).

* **LLM Factory and Clients:** (llm\_factory.py, llm\_google.py, llm\_local.py, possibly llm\_openai.py if they had it).  
  The LLM Factory decides which client to use based on environment or config. It could read an env var LLM\_PROVIDER (or even derive: if OPENAI\_API\_KEY is set use OpenAI, elif GOOGLE\_API\_KEY set use Google, elif a local model is available use that).

Each client module implements a generate\_answer(prompt: str) -> str.

* + llm\_google.py – uses the Google PaLM API as discussed, probably via the official google.generativeai if they used it, or via direct requests.
  + llm\_local.py – uses a local method. This might either call a local running server (like Ollama) or if they integrated something like the HuggingFace pipeline, they might have loaded a smaller model (though the mention of DeepSeek suggests a large one which is tough to run in code easily; likely expecting user to run it externally).

Possibly an llm\_openai.py existed if earlier steps used OpenAI. If they left it, maybe they still allow it. But since docs ask for Google/DeepSeek focus, they might have pivoted to those.

The factory pattern: For example,

class LLMFactory:

def \_\_init\_\_(self):

if config.GOOGLE\_API\_KEY:

self.client = GoogleLLMClient(config.GOOGLE\_API\_KEY)

elif config.OPENAI\_API\_KEY:

self.client = OpenAILLMClient(config.OPENAI\_API\_KEY)

elif config.LOCAL\_LLM\_ENABLED:

self.client = LocalLLMClient() # local might not need key, just host info

else:

raise RuntimeError("No LLM provider configured")

def generate(self, prompt: str) -> str:

return self.client.generate(prompt)

Then maybe instantiate one factory globally (like in main.py or config) as llm\_factory = LLMFactory() that can be imported where needed (though global singletons should be done carefully if containing state, but here it's stateless except config).

* **RAG Service / QA logic:** Perhaps rag\_service.py encapsulates the retrieval and generation process as a function answer\_question(question: str, db: Session). This would basically implement the steps I outlined in route, but outside the route for clarity. That would include:
  + Compute query embedding via embedding\_service
  + Query database for similar chunks
  + Build prompt
  + Call llm\_factory.generate
  + Possibly post-process the answer (like ensure it ends with a period, etc. or strip whitespace)

and returns the answer and maybe the references used so route can format them. Or it could directly return the AnswerResponse Pydantic model (the service could import the schema and instantiate it).

* **Misc Utilities:**
  + They might have used a library for environment config. Possibly the tutorial might have shown using python-dotenv to load .env in development. Or they just used Pydantic’s BaseSettings via FastAPI config.
  + Logging utility: maybe set up logging format and level globally.
  + If they needed to ensure Alembic migration runs at startup, they could call it in main but more common is to run externally. If in container, maybe entrypoint calls alembic upgrade head.

**Take note of BUILD\_CHECKLIST.md** as seen in repo listing. Possibly a document with steps to build environment. Not directly code, but an artifact in repo to ensure steps done.

**Integration among services:**  
The services are used by the routes. For example:

from services import rag\_service

@router.post("/query", response\_model=AnswerResponse)

def answer\_query(request: QueryRequest, db: Session = Depends(get\_db)):

answer, sources = rag\_service.answer\_question(request.question, db)

return AnswerResponse(answer=answer, sources=sources)

This keeps controller logic very clean.

The rag\_service in turn uses the lower-level services:

def answer\_question(question: str, db: Session):

q\_vec = embedding\_service.get\_embedding(question)

chunks = retrieval\_service.query\_top\_k(q\_vec, k=5, db=db)

prompt = prompt\_service.build\_prompt(question, chunks)

answer = llm\_factory.generate(prompt)

sources = [Source(document\_title=c.document.title) for c in chunks] # example

return answer, sources

(I added retrieval\_service and prompt\_service hypothetically; they might not split that granularly in code, but conceptually.)

**Check for concurrency**: If using async for I/O heavy tasks (embedding call, LLM call, file read), they could mark route as async def and call those external services with async (if libs support).

* cohere.Client.embed might not be async, so one would run it in threadpool with await run\_in\_threadpool(embedding\_service.get\_embedding, question).
* Similarly for DB query, since we used sync SQLAlchemy, that's in threadpool by FastAPI automatically.
* For LLM call, if it's a network request via requests, FastAPI will also not await it (requests is sync), so could do threadpool or use an async HTTP client (httpx).  
  But they may not have gone deep into async, possibly leaving as normal def (which FastAPI still runs in threadpool behind scenes).

**Testing the service layer**: Having logic in services means they can be unit tested by feeding dummy data (like a known vector and a mocked DB session or using an in-memory DB) and verifying output. It’s unclear if they wrote tests, but architecture supports it.

**Conclusion on services**: This layer is where the business logic of RAG lives. It transforms raw inputs to outputs through all the intermediate steps (embedding, searching, prompting, generation). The separation into multiple service modules indicates a clean architecture approach, improving maintainability. For instance, if tomorrow we switch to a different embedding model, we only change embedding\_service.py. Or if the prompt format should change, we edit prompt\_service.build\_prompt.

**6.5 Frontend Module Overview**

While the backend comprises the bulk of the unique logic, the frontend is what the end user interacts with. We provide an overview of how the frontend code is structured and how it works in conjunction with the backend.

**Location:** frontend/ directory. Inside this:

* Likely a package.json and package-lock.json specifying dependencies and scripts.
* A src/ folder with JavaScript/TypeScript code.
* If using React (common structure):
  + src/App.js or App.tsx which defines the main application component.
  + src/components/ maybe containing components like UploadForm, QuestionForm, AnswerDisplay, etc.
  + Possibly src/api.js or similar to abstract API calls (using fetch or axios).
* If using a specific framework like Next.js:
  + A pages/ directory with pages (like pages/index.js containing the main UI).
* If using Vue or Svelte:
  + src/App.vue or src/main.js which mounts the app.
  + But given typical course preferences, React seems most plausible.

**What the frontend does:**

* **File Upload UI:** A form or button where user selects a file. On submit, it calls the backend /documents/upload endpoint. It likely shows some loading indicator while upload goes on. After a successful upload (HTTP 200), it may display a message like "Document uploaded successfully" or add the document to a list of uploaded docs.
* **Question Input and Answer Display:** Perhaps a text input (maybe a form that on submit triggers an API call to /query). The front-end sends the question as JSON (e.g., { "question": "Your question?" }) to the backend. Meanwhile, it might display "Thinking..." to the user.  
  When the response comes back (with answer and sources), the front-end displays the answer text. If sources are provided, it might show them as clickable references or just as a list of document names used.
* Possibly it retains a history of Q&A in the UI (like a chat log). Some RAG apps do that to allow multi-turn (though our backend isn't specifically multi-turn-aware, but the user could follow up with another question – our system doesn't remember previous Qs by itself, but the user might refer implicitly to previous answer content which the system wouldn't know since it doesn't have conversation context retention).  
  Multi-turn could be simulated by including previous Q&A in context, but we haven't built that complexity (not mentioned, likely out-of-scope).
* **Error Handling:** If an API call fails (e.g., server down or returns error), the front-end should handle it gracefully. Maybe show an alert "Failed to get answer. Please try again."
* **Styling and Layout:** They likely kept it simple. Possibly using plain CSS or minimal library (if any, maybe a component library like Material-UI if React, or just basic Bootstrap classes).
* **State Management:** For a simple app, React's useState is enough to store say documentsList, question, answer, sources. If more complex, might use context or Redux, but likely unnecessary.

**Integration with backend:**

* The front-end needs to know where the backend is. If both served under same domain in production, it can use relative URLs (/documents/upload). In development, if front runs at localhost:3000 and backend at localhost:5000, they had to address CORS. Possibly they set proxy in package.json or configured in dev server (if CRA, adding "proxy": "http://localhost:5000" so that API calls automatically proxy to backend).
* Or they configured CORS in backend to accept localhost:3000.
* In the Docker environment, they might run both in the same network, possibly with backend at http://backend:5000 and front calling that (if served by an Nginx).
* There could also be environment in the front (like a config file or using environment variables with e.g. Create React App, one would have REACT\_APP\_API\_URL) and then in code:
* const API\_URL = process.env.REACT\_APP\_API\_URL || "";
* fetch(`${API\_URL}/query`, { ... })

In dev, .env might have REACT\_APP\_API\_URL=http://localhost:5000, in prod set accordingly.

**File structure specifics:**  
Maybe:

frontend/

├── src/

│ ├── index.js (entry point)

│ ├── App.js (main component with state logic)

│ ├── components/

│ │ ├── FileUploader.js

│ │ ├── QuestionForm.js

│ │ ├── AnswerDisplay.js

│ │ └── DocumentList.js (if they list files)

│ └── api.js (to centralize fetch calls, optional)

├── public/

│ └── index.html (container for the React app)

└── package.json

(This is if React. If Svelte, it would be a bit different but similar concept.)

**Interactive flow example:**

1. User opens the web app. Perhaps sees an upload area and below that a Q&A area (which might be disabled until at least one doc is uploaded).
2. User uploads "mydoc.pdf". The front reads the file into FormData and sends to /documents/upload. On success, the front might add "mydoc.pdf" to a list of docs on the UI or just notify success. The system now behind scenes has ingested it.
3. User types "What is the main topic of mydoc?" into the question box. Clicks ask. Front calls /query with JSON {"question": "..."}.
4. Front receives answer, say "The main topic is health benefits of green tea." It shows that in an answer area. It might also show "Source: mydoc.pdf".
5. User can then ask another question.  
   If the user asks something like "What about black tea?", the system currently has context only from mydoc (which might be about green tea). It might return "I don't know" or an unrelated answer because black tea wasn't in docs. The front should display whatever the backend returned (maybe an answer that says cannot find info if we coded such behavior).
6. The process repeats. If user uploads more docs, they can incorporate them.

**Edge on user experience:**  
If multiple docs are in the system, our backend currently searches across all chunks in DB. If user wanted to ask about a specific doc, maybe they'd mention doc name in question or we'd need to allow specifying doc. We didn't implement such filter in query (though one could by adding a doc\_id filter in the DB query or separate endpoint per doc). The UI might not give that level of control; presumably the user’s question will naturally contain terms relevant to one doc so the vector search will surface relevant info from the right doc.

**Conclusion on Frontend code:** The front-end is straightforward. It glues the user inputs to the backend services through HTTP. It ensures the user can utilize the system without needing to know curl commands or see JSON raw output. While not the primary focus of an AI project, it's an important component for demonstrating the project’s functionality in a user-friendly manner. The code in the front-end mainly deals with forms, event handlers for making fetch requests, and updating the DOM with results.

With the codebase structure explained, one can navigate the repository with understanding of what each part does:

* Models and database interactions (Section 6.2) underpin data persistence and retrieval.
* API routes (6.3) expose functionalities via endpoints and map requests to services.
* The service layer (6.4) implements the core pipeline logic (embedding, searching, generating).
* The front-end (6.5) provides the interactive UI.

In the next chapter, we will illustrate key processes in pseudocode form, consolidating these implementation details into clear step-by-step algorithms for document ingestion, vector generation/storage, and query answering.

**7. Main Processes – Pseudocode and Explanations**

This chapter breaks down the core processes of the system into pseudocode, providing a high-level but precise view of the algorithms and logic involved. We will cover three primary processes: (1) Document ingestion (from file upload to storage), (2) Vector generation and storage (how embeddings are created and saved), and (3) Query processing and answer generation (the end-to-end RAG query pipeline).

**7.1 Document Ingestion Pipeline**

*Goal:* Take an uploaded document file, extract its textual content, split into manageable chunks, compute embeddings for each chunk, and store everything in the database.

**Pseudocode:**

function ingest\_document(file):

# file is an UploadFile or similar object with attributes filename and file content

filename = file.filename

content\_bytes = file.read() # read file bytes (synchronously or asynchronously depending on environment)

# 1. Extract text from the file

if filename.endsWith(".pdf"):

text = extract\_text\_from\_pdf(content\_bytes)

elif filename.endsWith(".docx"):

text = extract\_text\_from\_docx(content\_bytes)

else:

text = content\_bytes.decode('utf-8', errors='ignore')

# (Note: extraction functions use appropriate libraries or logic.)

# 2. Split text into chunks for embedding

chunks = split\_text\_into\_chunks(text, max\_chars=500) # returns list of string chunks

# 3. Create a new document record in the database

document = Document(title=filename)

db.session.add(document)

db.session.commit() # obtain document.id after commit (assuming id is auto-generated)

# 4. For each chunk, generate embedding and store chunk record

for chunk\_text in chunks:

# 4a. Generate embedding vector for the chunk

vector = EmbeddingService.get\_embedding(chunk\_text)

# 4b. Create DocumentChunk record with reference to document

chunk\_record = DocumentChunk(document\_id=document.id, content=chunk\_text, embedding=vector)

db.session.add(chunk\_record)

end for

db.session.commit() # commit all chunks at once

return {"document\_id": document.id, "chunks\_stored": len(chunks)}

**Explanation:**

1. **File Reading:** The file content is read into memory. In an actual system, care should be taken with very large files (streaming instead of reading all at once), but for simplicity we show reading fully. The content bytes are then passed to an extraction routine based on file type.
2. **Text Extraction:** For PDFs, we might use a PDF parsing library to get text (pseudo extract\_text\_from\_pdf). For Word documents, a library like python-docx could extract text. If the file is plain text, just decode bytes. The extracted text is the full textual content of the document.
3. **Text Chunking:** Documents can be long, so split\_text\_into\_chunks breaks the text into smaller pieces. The pseudocode uses a simple heuristic of a max character length (500 chars) per chunk. A smarter implementation might break on sentence or paragraph boundaries for coherence. The result is a list of chunk strings (e.g., 5-10 sentences each). This ensures embeddings (which have input size limits) can handle them and that retrieval can pinpoint relevant parts.
4. **Database Insertion (Document):** A new Document record is inserted with a title (using filename or maybe a cleaned version of it). We commit to get an id because chunk records will reference it. (Alternatively, we could flush to get id without full commit in SQLAlchemy, but commit is straightforward here).
5. **Embedding and Storing Chunks:** For each chunk:
   * We compute its embedding by calling the EmbeddingService (which wraps the Cohere API call). This returns a vector (e.g., list of floats).
   * We create a DocumentChunk instance with a foreign key to the document, the chunk text, and the embedding vector. We add it to the session.

After processing all chunks, we commit once to save all chunk records. (Bulk commit is more efficient than committing each in loop.)

1. **Return result:** We return some info, e.g., the document ID and number of chunks stored, which can be used to confirm ingestion.

**Considerations & Enhancements:**

* We might parallelize embedding generation if we have many chunks to speed up ingestion (e.g., batch multiple chunks in one API call, which Cohere supports). This pseudocode does it sequentially. A more advanced version might do:
* for chunk\_batch in chunks in batches of 5:
* vectors = EmbeddingService.get\_embeddings(chunk\_batch) # batch call
* for each vector and corresponding chunk\_text:
* create chunk\_record with vector

This reduces number of API calls and can be much faster.

* Error handling: If extraction fails (corrupted file) or embedding API fails, we should handle exceptions (maybe rollback DB changes for that doc, and return an error response).
* In practice, document content might contain newline patterns, etc., that need cleaning (like multiple spaces, etc.); split\_text\_into\_chunks might implement trimming, removing excessive whitespace, etc.
* For very small documents, perhaps we don't need to chunk at all; our splitting function could return just one chunk if the text is shorter than max\_chars.
* The Document and DocumentChunk models likely also have timestamps (for auditing) but we skip that detail.
* If multiple users were involved (not mentioned in this project), we would associate document with a user id, etc., which the model and endpoints would handle. Not applicable here.

This ingestion process is invoked by the API route handling file uploads (as described in section 6.3). Once executed, the document’s content is prepared and ready for semantic search and QA.

**7.2 Vector Generation and Storage Process**

*Goal:* Compute embeddings for text data and store them for later similarity queries. This is partially covered in the above ingestion pseudocode (step 4), but here we focus on the embedding routine itself and how it's stored.

**Pseudocode for Embedding Generation Service:**

function get\_embedding(text):

# Uses Cohere API to get one embedding vector for the given text.

API\_KEY = config.COHERE\_API\_KEY

model = config.COHERE\_MODEL # e.g., "embed-english-v2.0"

# Prepare API request payload

payload = { "model": model, "texts": [text] }

headers = { "Authorization": f"Bearer {API\_KEY}" }

response = HTTP\_POST("https://api.cohere.ai/embed", headers, payload)

if response.status\_code != 200:

raise Exception("Embedding API call failed")

data = response.json()

embedding\_vector = data["embeddings"][0] # first (and only) embedding

return embedding\_vector # list of floats (e.g., length 768)

*(This pseudocode assumes a direct HTTP call for clarity. In actual code, one might use Cohere’s SDK, and handle exceptions accordingly.)*

**Explanation:**

* **Input:** A string of text (for instance, a chunk of a document or a query).
* **API Key and Model:** It uses a configured API key. We also specify which model to use. For consistency, the same model must be used for both documents and queries. The pseudocode expects a model name in config (which could default to a recommended model).
* **Payload Construction:** According to Cohere’s API, the embed endpoint expects a JSON body with a list of texts. Here we send a list containing our single text. We specify the model if required (some API versions might allow default model with just texts provided).
* **HTTP POST Request:** It sends the request to Cohere’s embedding endpoint. The actual URL and format might vary (Cohere’s docs show endpoints like /v1/embed or similar). We include the API key in the header for authentication (Cohere uses Bearer token).
* **Response Handling:** We check if the response is successful (status 200). If not, we throw an exception (in actual code we might log the error details or handle specific cases like rate limit).  
  If successful, the JSON contains an "embeddings" field which is an array of vectors (one per input text). We take the first vector (since we sent one text) and assign it to embedding\_vector.  
  This is likely a list of floats, e.g., [0.123, -0.456, ...] of length N (the dimension of the model, say 768).
* **Return:** The embedding vector is returned to the caller.

This function is used both in document ingestion (for each chunk) and in query processing (for the question). It encapsulates the details of calling the external API.

**Storage in Database:**

Once embedding\_vector is obtained, how is it stored in the database? The pseudocode in 7.1 shows creating a DocumentChunk with embedding=vector. Because the embedding column is of type VECTOR (pgvector), an ORM like SQLAlchemy will handle the conversion of a Python list to the database vector type. If using raw SQL, one would parameterize it such that the driver (psycopg2) knows how to bind it. Many drivers allow binding a list for a vector column directly.

In SQL explicitly, storing would look like:

INSERT INTO document\_chunks (document\_id, content, embedding)

VALUES (:doc\_id, :content, :embedding)

with embedding bound to the list of floats. The pgvector extension expects that list to match the declared dimension.

**Batch Embeddings (Optional):**

If we were to embed multiple pieces of text in one call:

function get\_embeddings(list\_of\_texts):

payload = { "model": model, "texts": list\_of\_texts }

headers = { "Authorization": f"Bearer {API\_KEY}" }

response = HTTP\_POST("https://api.cohere.ai/embed", headers, payload)

if response.status\_code != 200:

raise Exception("Embedding API call failed")

data = response.json()

embeddings = data["embeddings"] # list of vectors, one per input

return embeddings

We could then iterate through returned vectors to create chunk records. This is more efficient (fewer network calls) and Cohere likely allows a batch of up to some limit (maybe 96 texts as per docs). Our ingestion could use this to reduce overhead significantly.

**Memory / Performance:**

* Converting large text to embeddings might have restrictions (some APIs cut text beyond certain token count). Possibly our split\_text\_into\_chunks ensures chunks are within that limit. If not, Cohere may truncate or error. Ideally we ensure chunk size is under model's max token (like 512 tokens).
* Storing many embeddings: They are high-dimensional floats, but Postgres handles that (with some overhead in storage). For example, 768 floats (4 bytes each) is ~3KB. If a doc is split into 20 chunks, that's 60KB of vector data plus text. This is fine for typical usage. Searching through them via index is efficient due to pgvector's ANN index.

**Edge Cases:**

* If text is empty or just whitespace, what do we do? Possibly skip embedding to avoid waste, or embed and get some vector (Cohere likely returns a neutral vector). We might choose to skip storing empty chunks. Our split\_text\_into\_chunks might not produce empty chunks anyway.
* If the Cohere service is down or the API key is invalid, our function raises an exception. The route handling ingestion or query should catch it and return an HTTP error (like 500 or a custom message "Embedding service unavailable").
* Over time, if we wanted to change the embedding model (say a new model with larger dimension), we should re-embed existing docs for consistency. If not, vectors in DB are from old model, queries from new might not align. So it's crucial to maintain the same model or plan re-indexing if changed. This is more of an operational note.

In summary, the vector generation function (get\_embedding) is a small but crucial piece. It turns language to math, enabling all subsequent similarity computations. It is abstracted so that if we want to switch from Cohere to another provider, we could modify this function without changing the rest of the pipeline.

**7.3 Query Processing and Answer Generation**

*Goal:* Given a user’s question, find relevant information from stored documents and generate an answer using an LLM, returning the answer (and references) to the user.

**Pseudocode:**

function answer\_query(question):

# 1. Embed the user's question into a vector

q\_vector = EmbeddingService.get\_embedding(question)

# 2. Retrieve top-K similar chunks from the database

K = 5

results = db.query(DocumentChunk)

.order\_by(vector\_distance(DocumentChunk.embedding, q\_vector))

.limit(K)

.all()

# results is a list of DocumentChunk objects (or records) sorted by similarity

if results is empty:

answer\_text = "I'm sorry, I could not find relevant information in the documents."

return answer\_text, [] # no sources

# 3. Build the LLM prompt with retrieved context

prompt = "Use the following document excerpts to answer the question.\n"

for i, chunk in enumerate(results, start=1):

prompt += f"[{i}] {chunk.content}\n"

prompt += f"\nQuestion: {question}\nAnswer:"

# 4. Generate answer using the LLM

answer\_text = LLMFactory.generate(prompt)

# (LLMFactory chooses Google or local model based on config, as described earlier)

# 5. Prepare sources list

sources = []

for chunk in results:

source\_info = { "document": chunk.document.title }

# optionally include more info like page number or snippet

sources.append(source\_info)

return answer\_text, sources

**Explanation:**

1. **Embedding the Query:** The question (in natural language) is converted to an embedding vector q\_vector using the same EmbeddingService. This places the question in the same vector space as document chunks.
2. **Similarity Search in Database:** We use the q\_vector to find the nearest neighbor chunks:
   * The pseudocode uses a placeholder function vector\_distance to denote the similarity metric. In practice, if using pgvector with cosine, it might be embedding <-> q\_vector in raw SQL or some function provided by SQLAlchemy. We retrieve the top K (here 5) chunks.
   * These results include the chunk text and a reference to the document it came from (thanks to our ORM relationships or by joining Document table if needed).
   * If the results list is empty (meaning no documents at all in DB or the search failed to find any because DB was empty), we handle that by giving an apologetic answer indicating no info. This prevents passing an empty context to LLM, which might cause it to hallucinate an answer with no basis.
3. **Building the Prompt:** We construct a textual prompt for the LLM that contains:
   * The instruction that it should use the given excerpts to answer.
   * Each retrieved chunk is enumerated and included, prefixed by an identifier [1], [2], ...
   * We then append the actual user question and "Answer:" to prompt the model to output an answer. This format encourages the model to refer to the [1] [2] content.

We have separated context and question clearly. The design "Use the following document excerpts..." guides the model. This method is known to reduce hallucinations by explicitly telling it to ground answer on context.

1. **LLM Generation:** The LLMFactory.generate is called with the prompt. Under the hood, this will:
   * If using Google PaLM: call the API and get the answer text.
   * If using DeepSeek/local: send to local model and get answer text.  
     The model sees something like:
2. [1] ...content of chunk1...
3. [2] ...content of chunk2...
4. ...
5. Question: What are catechins and how do they benefit health?
6. Answer:

and then it will produce something along the lines of: "Catechins are natural antioxidants found in green tea (as seen in [1]). They benefit health by boosting metabolism and reducing inflammation, as the document suggests."  
The content in parentheses might not actually appear unless model is specifically tuned to cite [1]. Some models might spontaneously cite [1] since it saw them in input. But we did not explicitly ask it to produce bracket citations. If we wanted that, we might add to the prompt "Answer with references like [1] or [2] for the facts used."

In the pseudocode, we kept prompt simpler, so the model might just answer with the info but not mention the numbers.

1. **Preparing sources list:** We want to return to the user which documents (or chunks) the answer drew from. In our simple approach, we gather the document titles of each chunk in results. We create a list of source info dictionaries. For now, just document title (like "green\_tea.pdf"). Optionally, one could include the snippet or chunk id.  
   If the front-end is just going to display "Source: green\_tea.pdf, research.txt", this suffices. If we had more time, we could filter duplicates (if multiple top chunks from same doc, better list document once).  
   Or even map the chunk indices to the document name for an inline reference scheme (i.e., if answer had [1], we could present [1]: "green\_tea.pdf").  
   For brevity, sources each have document title.
2. **Return:** The function returns the answer text and the sources list. The API route will then put these into a response model (AnswerResponse with those fields). This gets serialized to JSON to the client.

**Edge Cases & Potential Improvements:**

* If the top retrieved chunks are not actually very relevant (maybe the question is slightly off from what's in docs), the answer might be inaccurate or generic. We could implement a threshold: if the similarity score of the top result is below some value, maybe we say "no relevant info found." But that requires computing the distance. pgvector can provide the distance (we could fetch the distance by doing something like SELECT embedding <-> :qvec as distance along with chunk). With ORM, one might not do threshold easily without raw SQL.  
  For simplicity, we didn't add threshold; the model likely can handle if context is irrelevant by either saying "I don't know" or making something up (hallucination risk). We somewhat mitigated by telling it "use the excerpts", so if excerpts are irrelevant it might say "The excerpts do not provide information on that."
* Multi-turn context: As is, each query is independent (stateless). If the user asks a follow-up that references previous answer, our system wouldn't know previous answer's content unless the user repeats it. Some RAG systems append a conversation history to context, but that wasn't part of requirements. So not implemented.
* LLM errors: If the LLM API fails (maybe network error), we should catch it and possibly return an error message or try a fallback. We didn't show it to keep pseudocode concise. But a robust system would wrap LLMFactory.generate in try/except and handle exceptions similarly to embedding failures.
* Performance: The DB query is fast given index. The embedding call is quick (a few hundred ms). The LLM call might be the slowest (a couple seconds perhaps if using a large model or local model). We might consider asynchronous parallel retrieval and LLM call if possible (like start LLM call and retrieval simultaneously, but you need retrieval results for prompt, so not parallelizable in straightforward way).  
  But one could embed query and do retrieval concurrently with building part of prompt, though practically, retrieval is so fast it doesn't bottleneck.
* Overlap in chunks: If top results are adjacent chunks from same doc, the answer might benefit if those chunks were merged. We did not handle that, but it's a nuance. Possibly not needed if the question matches one chunk mostly.

**Example to illustrate:**

Documents contain text about green tea and black tea in separate chunks. User asks "Does green tea contain antioxidants and what do they do?"

* Perhaps chunk [1] is "Green tea contains catechins, a type of antioxidant."
* Chunk [2] is "Catechins help improve metabolism and reduce the risk of diseases."

Our retrieval likely picks those two. We prompt:

[1] Green tea contains catechins, a type of antioxidant.

[2] Catechins help improve metabolism and reduce the risk of diseases.

Question: Does green tea contain antioxidants and what do they do?

Answer:

The LLM sees [1] answers first part and [2] answers second part. It might produce:  
"Yes, green tea contains antioxidants called catechins. These catechins improve metabolism and reduce the risk of certain diseases."  
That is a good answer derived from context.

We would return that with sources = [ {"document": "tea\_info.pdf"}, {"document": "tea\_health.txt"} ] (assuming chunk1 came from one doc, chunk2 from another). If both from same doc, source list might be duplicate doc, we could simplify that. Possibly we could unify sources by document, but pseudocode above doesn't do that (it appends each chunk's doc, which could have duplicates).

A refined approach:

source\_docs = set(chunk.document.title for chunk in results)

sources = [ {"document": title} for title in source\_docs ]

This would unique them.

**Returning to user:** The front-end can take answer\_text and show it, and list sources. The user now has an answer and some trace of origin (in case they want to verify or read more from the actual docs).

The RAG pipeline is thus completed in these steps. This pseudocode aligns with what we described in Chapter 4 system architecture and Chapter 6 code structure.

Finally, we ensure that this pseudocode captures the essence of the implemented logic, albeit in a simplified form, and it can serve as a reference for understanding and verifying the actual code.

By following these algorithms, one could reimplement or debug the project’s main operations, or modify it (for instance, to change the retrieval strategy or the prompting approach).

With the process breakdown covered, we have essentially described how the system works from data ingestion to answer output. Next, we will discuss project planning aspects like timeline and team contributions, before moving on to architecture diagrams, future work, and appendices.

**8. Project Planning and Management**

Developing a project of this scope required careful planning, team coordination, and iterative problem-solving. In this chapter, we outline how the project was planned and executed, highlighting the timeline of milestones, roles of team members, and challenges encountered (along with how we addressed them).

**8.1 Development Timeline and Milestones**

The project was structured into a series of phases, each corresponding to key features or components of the system. This phased approach was influenced by the educational “step-by-step” nature of the original tutorial (the mini-rag course) and adapted to our project’s goals. Below is the timeline of milestones:

* **Week 1-2: Project Initialization and Architecture Design**  
  *Milestone:* Define project objectives, study RAG architecture, select technology stack.  
  *Actions:*
  + Set up version control (GitHub repository).
  + Write initial Project Requirement Document (PRD) outlining scope (the repository contains PRD.txt which helped define the problem and solution approach).
  + Decide on using FastAPI, PostgreSQL/pgvector, and identify integration points for Cohere and LLMs (OpenAI/Google/DeepSeek).
  + Prepare initial data (sample documents) for testing assumptions.
  + Outcome: A high-level architecture diagram and plan (refined in section 4 and used to guide development).
* **Week 3: Backend Framework and Basic API Setup**  
  *Milestone:* Implement basic FastAPI server structure with dummy endpoints.  
  *Actions:*
  + Create the FastAPI app and set up main.py.
  + Implement a simple health check endpoint to verify server running.
  + Integrate Pydantic for config (e.g., loading API keys from .env).
  + If multiple team members, Backend Developer would lead this with input from others.
  + Outcome: Running FastAPI app with trivial endpoints, Dockerfile base created.
* **Week 4: File Upload and Document Storage**  
  *Milestone:* File ingestion endpoint working, storing files content to DB.  
  *Actions:*
  + Set up database models for Document and DocumentChunk (initially without vector, just text).
  + Implement file upload route (POST /documents/upload).
  + Integrate a file parser for PDFs and text (ensuring we can extract content, possibly by testing with known PDF).
  + Save raw text to DB (this week possibly skipping embedding to get fundamentals working).
  + Outcome: We could upload a file and see its text saved in the database (e.g., as one chunk for now). Verified by retrieving via a temporary endpoint or DB query.
* **Week 5: Database Integration and Vector Setup**  
  *Milestone:* Enable pgvector in Postgres and store embeddings for document chunks.  
  *Actions:*
  + Install pgvector extension on development Postgres, update DB models to include embedding field (e.g., using SQLAlchemy’s Vector type).
  + Write migration script (via Alembic) to add vector column.
  + Integrate Cohere’s embedding API: implement EmbeddingService and test it on sample text (ensuring API key works, etc.).
  + Modify ingestion process to chunk text and generate/store embeddings for each chunk.
  + Outcome: After uploading a file, its content is stored in chunks with embeddings. Verified by checking the DB (embedding column populated) and that the service can handle multiple chunks.
* **Week 6: Query Endpoint and Retrieval Logic**  
  *Milestone:* Implement semantic search and question-answering route with dummy LLM response.  
  *Actions:*
  + Write the query route (POST /query) skeleton: accept question, embed it, query the DB for similar chunks.
  + For now, instead of calling an LLM, just return the top chunk’s text as an "answer" or a placeholder answer like “Found relevant info in Document X”.
  + This was to test the retrieval pipeline end-to-end. We tested by uploading a doc with known content, then querying a term from it and seeing if the route returns that chunk/snippet.
  + Team roles: Data Engineer ensures vector search returns correct chunk, Back-end dev ensures FastAPI properly returns output.
  + Outcome: Successful retrieval of relevant content from DB for given query. Answer format still placeholder but shows pipeline working.
* **Week 7: LLM Integration (OpenAI/Google)**  
  *Milestone:* Connect an actual LLM to generate answers from retrieved context.  
  *Actions:*
  + Choose initial LLM provider (OpenAI’s GPT-3 was likely used first, given known API patterns; then extended to Google PaLM).
  + Implement LLMFactory and OpenAI client (e.g., using openai library) to call GPT-3 with the constructed prompt.
  + Test query end-to-end: ask a question, retrieve chunks, build prompt, get answer from GPT-3. Evaluate correctness of answers.
  + If authorized, also test with Google’s model by switching config (this required setting up GCP project and API key).
  + Outcome: The system can return actual answers. For example, after uploading a document about green tea, asking "What are benefits of green tea?" returns a coherent answer citing antioxidants etc., rather than a static snippet. This was a big milestone showing RAG working with generative AI.
* **Week 8: Local LLM Integration (DeepSeek via Ollama)**  
  *Milestone:* Demonstrate the system working with a local model, achieving data privacy.  
  *Actions:*
  + Set up environment for running DeepSeek 7B (perhaps using provided Colab+Ngrok method for a quick test as per README).
  + Implement the LocalLLMClient in LLMFactory to send prompt to the local model’s API.
  + Test using a smaller query and context due to local model resource limits.
  + This was perhaps done as a proof-of-concept rather than for heavy usage, given hardware constraints in development.
  + Outcome: Verified that if configured, the system can use a local LLM (DeepSeek) to answer queries with acceptable output. This milestone was important for the *academic demonstration* of flexibility, even if for production one might stick to cloud LLM until local hardware is sufficient.
* **Week 9: Frontend Development**  
  *Milestone:* Build a simple web interface for file upload and Q&A interaction.  
  *Actions:*
  + Frontend Developer sets up a React app (or chosen framework).
  + Implement file upload component that calls backend.
  + Implement question input and display of answer and sources.
  + Ensure CORS or proxy is configured so front can talk to back in dev mode.
  + Style the UI to be presentable (using basic CSS or a UI kit).
  + Outcome: A functional UI where one can choose a file, upload it, then enter questions and see answers. This made the project user-friendly and demo-ready.
* **Week 10: Testing, Evaluation and Refinement**  
  *Milestone:* Test the entire system with various documents and questions, and improve accuracy/robustness.  
  *Actions:*
  + Conduct test cases: e.g., upload a sample PDF on a known topic, ask relevant and irrelevant questions. Evaluate answer correctness and whether source referencing works.
  + Tune the prompt if needed: if we noticed the LLM not using context properly or hallucinating, we tweaked the prompt wording (e.g., adding "if answer not in context, say you don't know").
  + Address edge cases: what if two files with similar info are uploaded? Check that it still handles sources and doesn't mix them up.
  + Implement any missing features: e.g., if we found duplicate sources listing, we might unify them. If we realized we need a list-documents API to show uploads in UI, add it.
  + Fix bugs: e.g., memory leak issues, handling of large files (maybe limit file size).
  + Outcome: A refined system with better reliability. At this point the main functionality was complete.
* **Week 11: Deployment Setup and Documentation**  
  *Milestone:* Prepare for deployment and create documentation (user guide and technical report).  
  *Actions:*
  + Finalize Docker configurations for smooth deployment. Test running entire stack with docker-compose on a staging server or local environment.
  + Write user instructions (perhaps a README update) on how to run the system and use the interface.
  + Compose comprehensive documentation (this document) covering theoretical background and system details. This took significant time as we compiled citations and ensured clarity.
  + Each team member contributed sections related to their expertise (e.g., an ML specialist wrote theoretical background on transformers and embeddings, a back-end specialist wrote about architecture and code).
  + Outcome: A fully containerized application ready to be deployed (on a cloud VM or possibly a PaaS) and a polished documentation to accompany the final submission/presentation.
* **Week 12: Final Testing and Presentation**  
  *Milestone:* Conduct final end-to-end tests and present the project to evaluators.  
  *Actions:*
  + Run the system in an environment similar to where it will be demonstrated (maybe a laptop or server), to ensure no configuration is missing.
  + Prepare a demo script: decide which document to upload and what questions to ask during the presentation to showcase capabilities (maybe a small dataset of documents was prepared, like some Wikipedia articles or technical docs, to use for demonstration).
  + Each team member rehearsed explaining their portion (the introduction, the demo, the technical details Q&A likely to follow).
  + If any minor issues were found (like a certain file format causing error), either fix or plan to avoid that scenario in demo.
  + Outcome: The team is confident in the project’s functionality and prepared to deliver a structured presentation highlighting objectives, methods, and results.

The above timeline highlights that initial weeks focused on foundation (architecture, database, upload), middle weeks integrated the intelligent components (embeddings, LLMs), and later weeks polished the user interface and reliability. Aligning with academic timelines, this might have spanned roughly 3 months (which fits a typical one-semester project schedule).

**8.2 Team Roles and Contributions**

The project was developed by a team, with each member taking on specific roles in line with their expertise. Below are the typical roles and how contributions were divided:

* **Project Manager / Team Lead:**  
  *Responsibilities:* Overseeing project progress, ensuring milestones are met on schedule, and coordinating between team members. Also acted as a liaison with stakeholders (e.g., academic supervisors) with regular updates.  
  *Contributions:* The PM developed the initial project plan, arranged team meetings, and tracked tasks in a simple issue tracker. They also contributed to high-level design decisions (e.g., deciding to integrate pgvector and which LLM to use when) and reviewed code to ensure consistency. Additionally, they compiled the final documentation (gathering sections from others and integrating them into one voice).
* **Backend Developer:**  
  *Responsibilities:* Implementing the server-side logic, including API endpoints, database interactions, and integration of external services (embedding API, LLM APIs).  
  *Contributions:* The backend dev wrote most of the code in src/routes and src/services. They set up the FastAPI app structure, wrote the ingestion and query endpoints, and ensured the ORM models and Alembic migrations were correct. They also handled the Dockerization of the backend service. This person troubleshooted issues like CORS, database connection pooling, etc., during development. They wrote unit tests for service functions to verify that retrieval returns expected chunks and that the LLMFactory chooses the correct client based on config.
* **Machine Learning Engineer / NLP Specialist:**  
  *Responsibilities:* Handling aspects related to language processing, such as the embedding model selection, vector database tuning, and prompting strategy for the LLM.  
  *Contributions:* The ML specialist researched the best practices for RAG (e.g., read Cohere’s docs on RAG and literature like the RAG paper to guide approach). They fine-tuned the text splitting logic to ensure semantic coherence in chunks. They also worked on prompt engineering: experimenting with different prompt phrasing to get the best results from the LLM (like whether to include "cite the source" etc.). Furthermore, they evaluated output quality: after initial LLM integration, they analyzed cases where the answer was wrong or too verbose and adjusted temperature or prompt accordingly. This member also suggested the use of DeepSeek for local inference after noticing trending open-source models, thus adding an innovation aspect to the project.
* **Frontend Developer:**  
  *Responsibilities:* Creating the user interface for the application and ensuring a smooth user experience.  
  *Contributions:* The frontend dev built the React (or chosen framework) application, implementing components for uploading files and asking questions. They styled the UI to be clear and accessible. This person also handled connecting the frontend to the backend: configuring proxy or CORS, handling responses (displaying answers and source citations neatly). They performed user testing with colleagues to improve the layout (for example, ensuring that when an answer comes in, it scrolls into view properly, etc.). They also integrated error handling on the client-side (like showing a message if the server is down or returns an error). If the project was small team (2-3 people), this role might have been combined with Project Manager or Backend, but in any case, someone took charge of the UI.
* **DevOps / Deployment Engineer:**  
  *Responsibilities:* Setting up the deployment environment, Docker configuration, and ensuring all services (DB, backend, frontend) run together.  
  *Contributions:* This role might have been part of the backend dev’s responsibilities in our team. Key tasks included writing the docker-compose.yml, configuring environment variables for production, and ensuring the Postgres service had persistence and that pgvector extension is created (they wrote instructions or scripts to run CREATE EXTENSION vector). They also possibly set up a staging environment on a cloud VM (for example, using DigitalOcean or AWS Lightsail) to test real-world deployment. They handled issues like adjusting UVicorn workers for production (maybe using Gunicorn with Uvicorn workers for better concurrency in deployment). Finally, they prepared the system for demonstration by pre-loading some example documents if needed (to avoid waiting for an upload during presentation, depending on strategy).

All team members collaborated in testing and debugging. For instance:

* The backend dev and ML engineer worked closely to fix issues in retrieval (like ensuring the correct similarity metric was used).
* The ML engineer and backend dev paired to integrate the Cohere API (checking that the data returned by Cohere is correctly inserted into the DB).
* The frontend dev often coordinated with the backend dev to adjust API responses (for example, if the frontend needed the sources in a certain format, they communicated and the backend schema was adjusted accordingly).

Regular meetings (weekly or twice a week) were held to synchronize progress. In these, each member reported what they did, any blockers, and planned next tasks. The PM facilitated these and updated the task board accordingly.

By dividing roles, the team ensured parallel progress: while the backend was being built, the frontend could be developed with mock data; while ML aspects were researched, the basic infrastructure was being coded. This concurrency in development allowed us to finish within the allotted time.

**8.3 Challenges Faced and Solutions Implemented**

During the course of the project, we encountered several challenges. We describe the major ones here and how we addressed them:

* **Challenge 1: Extracting Text from Various Document Formats**  
  **Issue:** We needed to support at least PDF files (and optionally Word documents) to ingest content. PDFs can be complex (multi-column, images, etc.), and parsing them reliably is non-trivial. Initially, using a basic PDF-to-text tool yielded text but with odd line breaks and artifacts, affecting chunking and possibly embeddings.  
  **Solution:** We integrated a well-known PDF parsing library (PyMuPDF a.k.a fitz). This library allowed us to extract plain text per page effectively. We then concatenated pages. We also performed post-processing on the extracted text: removing hyphenation (where a word is split across lines) and combining lines of paragraphs. This made the text more coherent for our chunking algorithm. For .docx, we used python-docx which gave text but had some extra newlines; we stripped those appropriately. We established a fallback that if any format fails, we log a warning and attempt to treat the file as plain text (this worked e.g. for .txt files natively). Testing with a variety of PDF layouts helped ensure our extraction was robust. We documented supported formats and recommended users to provide text-based PDFs (scanned PDFs without OCR would yield no text – we noted that as a limitation to not focus on implementing OCR in this project).
* **Challenge 2: Tuning Vector Similarity Search Performance**  
  **Issue:** With pgvector, we needed to ensure queries were fast, especially as the number of document chunks grows. Initially, we inserted data and queries were doing a sequential scan (because we hadn’t created the index yet, or because our query might not have been written to use it). This was slow with a few thousand vectors.  
  **Solution:** We created an IVFFLAT index on the vector column with appropriate parameters. We chose lists = 100 (an index tuning parameter) after some reading; this was a good trade-off for our dataset size (hundreds to low-thousands of chunks). Once the index was in place, query times dropped significantly (from maybe 200ms to 5-10ms for the vector similarity part). Another hiccup was getting SQLAlchemy to use the <-> operator. We solved it by using the vector\_cosine\_ops in the index and writing raw SQL for the ORDER BY in our query (since SQLAlchemy’s support was limited at the time). We also tested the search with and without filtering by document and found it fine to search globally as our volume is not huge. For future scale, we considered adding a filter by document categories if needed (not implemented since out-of-scope but we left notes in code). This challenge taught us about database-specific optimization and verifying that our ORMs actually utilize indexes (we used EXPLAIN ANALYZE on our queries via a DB tool to confirm index usage).
* **Challenge 3: Ensuring Relevant Context and Preventing LLM Hallucinations**  
  **Issue:** In early tests, we observed that if the retrieved chunks only partially cover the answer, the LLM sometimes produced extra information not in the docs (hallucinating) or answered in very general terms. For example, when asked a specific detail, the model might give a plausible-sounding answer that wasn't actually supported by the provided excerpts. This is a known challenge in RAG systems.  
  **Solution:** We approached this on multiple fronts:
  + **Prompt Refinement:** We modified the prompt to explicitly instruct the model to use the given documents. We also experimented with adding a line like "If the documents do not contain the answer, respond that you don't know." This reduced hallucinations by making the model cautious. We had to be careful: sometimes the model would then say "I don't know" even when answer was present but required synthesis. After tweaking, we settled on a prompt (as shown in pseudocode) that strongly implies the answer is in the excerpts. This improved factuality.
  + **Limiting LLM Creativity:** We set the LLM generation parameters to be more conservative (e.g., temperature = 0 or 0.2 for deterministic or less varied output). This way, it sticks closer to given info and doesn't invent as much. For critical queries, a temperature of 0 was ideal as it essentially does a greedy deterministic generation which often means it uses the prompt content verbatim if relevant.
  + **Increasing Context Quality:** We decided to fetch top 5 chunks instead of top 3 initially, for example, to ensure enough context is given. However, too many can confuse the model. We found 5 to be a reasonable number. We also adjusted chunk size; initially, chunks of 1000 characters were used but sometimes that included extraneous info. Splitting into ~500 char chunks (or splitting by semantic boundaries) meant retrieved chunks were tightly relevant or single-fact chunks. This precision helped the model focus.
  + **Testing edge queries:** We tried deliberately tricky queries, e.g., asking something not in documents to see how the model responds. With our instructions, the model often answered with a cautious "The documents provided do not mention that." which is acceptable. Without the instructions, the model was more likely to make something up to answer anyway. So our solution was effective, though not foolproof. Extremely advanced models (like GPT-4) hallucinate less, but we primarily had access to GPT-3.5-turbo and Google’s text-bison; these improvements made those models behave better.
* **Challenge 4: Integration of Multiple LLM Providers**  
  **Issue:** We wanted to support both a cloud API (Google) and a local model (DeepSeek). Managing two different interfaces and testing both was challenging, especially since the local model required environment setup that was different (and heavy).  
  **Solution:** We designed the LLMFactory with a clear interface so either backend could be plugged in. We used environment flags to easily switch. We developed using OpenAI (since it was fastest to test) and then swapped to Google for final runs, adjusting minor differences (e.g., Google’s API might require a different prompt structure or had a max length lower, etc.). For DeepSeek, since we could not run the 67B model on our dev machines, we used the Google Colab approach provided by the tutorial: essentially running the model in Colab (with a smaller 7B variant due to Colab limits) and exposing it via an API. This itself was complex, but we treated it as an external API call (just like calling Cohere or Google). We ran a few queries through it to demonstrate it works (though its quality was a bit lower than the big models, as expected). The key to integration was abstracting the differences: e.g., OpenAI returns response.choices[0].text, Google returns response.result, DeepSeek server returned a JSON with 'generated\_text'. Our LLMClient.generate functions all normalize to returning just the text string. By doing this, the rest of the code (the query endpoint logic) didn't need to care which one was used.
  + We documented how to switch providers, and tested each separately. The time overhead spent on multi-provider support was non-trivial but valuable for the project demonstration of flexibility.
* **Challenge 5: Handling Large Documents and Memory Constraints**  
  **Issue:** Some test documents were quite large (e.g., a 50-page PDF). Ingestion of these took a long time and large memory (reading entire PDF, generating maybe hundreds of embeddings). Also, if multiple such documents were uploaded, our vector DB could have a few thousand vectors, which is fine, but generating them on the fly took a while (and the web UI might time out on upload).  
  **Solution:** We implemented a few mitigations:
  + We set a file size limit for uploads (e.g., 5MB) and noted it in the UI (the frontend checks file size and warns if too large). This prevents extremely large content ingestion that we can't handle in real-time.
  + We offloaded the embedding generation to background tasks for very large docs. Specifically, after extracting text and splitting, if chunk count was huge (say > 100), we returned a response like "Document is being indexed, please wait a moment before querying." and ran the embedding & DB insert in a background thread (using FastAPI's BackgroundTasks). This way the HTTP request didn't timeout. This was an optional improvement; for moderate docs, it just does it inline.
  + We also considered and documented that for truly large knowledge bases, one should use a more robust pipeline (like asynchronous indexing or using a message queue). For our scope, these adjustments sufficed.
  + Memory-wise, we made sure to delete or nullify large variables when not needed (like after parsing text and splitting to chunks, we could drop the full text to free memory). The Python garbage collector would handle it, but in some instances explicitly clearing large lists (del content\_bytes after done) helped peak memory usage.
* **Challenge 6: Team Coordination and Merge Conflicts**  
  **Issue:** With multiple team members working concurrently (particularly on overlapping areas like the query logic which involves both backend and ML concerns), we occasionally encountered merge conflicts in Git or duplicated work.  
  **Solution:** We adopted a simple but effective branching strategy: each major feature was developed in its own Git branch (e.g., feature/frontend-ui, feature/embedding-integration, feature/google-llm). We merged into a develop branch after code review by at least one other person, then eventually into main. We also communicated frequently on who is working on what to minimize overlap. For example, when the ML engineer was adjusting the prompt format, the backend dev paused work on that part to avoid collision. We also used code commenting and documentation to clarify tricky sections (like why we set a particular parameter for the LLM). By the end of the project, our commit history was organized, which made writing the final documentation easier (we could trace why certain decisions were made by looking at commit messages referencing issues/challenges).

Through these challenges, the team learned a lot: from low-level technical fixes (like database indexing) to high-level AI behavior tuning (like prompt engineering). Each challenge overcame made the system more robust and the team more skilled in building complex AI applications.

In conclusion, strong planning and teamwork allowed us to navigate the typical hurdles of an AI integration project. We maintained focus on the core goals (building a functional, accurate RAG system) while also fulfilling additional objectives like multi-LLM support and user-friendliness. The lessons from these challenges will certainly inform our future projects and have been documented to aid others who work on similar systems.

**9. Visual Diagrams: Architecture and Data Flow**

To complement our textual description, this chapter presents visual diagrams that illustrate the system architecture and the data flow through the Retrieval-Augmented Generation (RAG) pipeline. These diagrams serve as a quick reference to understand how components interact and how information moves from input to output.

**9.1 System Architecture Diagram**

*Figure 9.1: System Architecture Overview.*  
*This diagram depicts the high-level architecture of the Local RAG system. The user interacts through a web frontend (browser), which communicates with the FastAPI backend. The backend consists of various services: the Document Ingestion module (handling file parsing and chunking), the Embedding Service (calling Cohere API to produce vectors), the Vector Database (PostgreSQL with pgvector) storing document embeddings, and the LLM Integration module (which interfaces with external Large Language Models like Google PaLM API or a local DeepSeek model). The arrows indicate the flow of data: users upload documents which are processed and stored, and when a query is asked, the system retrieves relevant vectors from the database and sends the compiled context to the LLM to generate an answer.*

In Figure 9.1, notice how the components are arranged:

* The **User Browser** (left) sends HTTP requests for uploading files or asking questions.
* The **FastAPI Backend** (center) is composed of several sub-components:
  + **Upload & Chunking**: receives documents, extracts text, splits into chunks.
  + **Embedding Service**: for each chunk or query, calls **Cohere API** (top) to get embeddings.
  + **PostgreSQL (with pgvector)**: stores the text chunks and embeddings in a persistent store.
  + **Retrieval & RAG Orchestrator**: when a query comes, it embeds the query, uses pgvector to find similar chunks, then constructs a prompt.
  + **LLM Connector**: sends the prompt to an LLM. The diagram shows two possible paths: one to **Google PaLM API** (external service on the right, representing cloud LLM) and one to **Local DeepSeek Model** (external but could be on-premise, shown at bottom-right). Only one is used at a time depending on configuration.
* The **Answer** flows back from the LLM to the backend, which then responds to the user’s browser with the answer (and sources).

This architecture ensures modularity: each external dependency (Cohere, LLMs) is abstracted behind service interfaces, and the database decouples storage from processing.

**9.2 Data Flow Diagram (RAG Pipeline)**

To detail the dynamic behavior, Figure 9.2 illustrates the data flow through the system when a user asks a question:

*Figure 9.2: Data Flow in Retrieval-Augmented Generation Pipeline.*  
*(Adapted from NVIDIA’s RAG pipeline concept). The sequence is numbered: (1) The user question is sent to the backend. (2) The backend generates an embedding of the question using Cohere (converting it to a query vector). (3) This query vector is used to perform a similarity search in the pgvector database, retrieving the top relevant document chunks (shown as colored documents in the vector database icon). (4) The retrieved text chunks are concatenated into a context which is combined with the question to form a prompt. (5) This prompt is fed into the LLM (Google PaLM or DeepSeek model), which generates an answer. (6) The backend returns this answer to the user, often with citations referencing the documents used.*

Following the numbers in the diagram:

1. **User Query:** A question like "What are the health benefits of green tea?" enters the system.
2. **Query Embedding:** FastAPI backend calls Cohere to produce an embedding vector (represented by the small orange circle vector).
3. **Vector Search:** That vector is compared against the stored vectors in Postgres/pgvector. The diagram shows some document icons in the vector DB; the most similar ones (highlighted) are retrieved.
4. **Context Construction:** The text from those relevant documents (or excerpts) is pulled and packaged together with the question. In the diagram, you can see the prompt being assembled (question + excerpts).
5. **LLM Processing:** The prompt goes to the chosen LLM. The LLM icon in the diagram shows it reasoning with both the question and documents to produce an answer. Notably, it can incorporate facts from the provided context (minimizing reliance on its internal memory).
6. **Response to User:** The answer (for example, "Green tea contains antioxidants (catechins) that boost metabolism and reduce disease risk.") is sent back to the user's browser. The diagram indicates this with a reply icon and optionally cites Document A and B as sources.

The above data flow emphasizes how external knowledge (the documents) is injected into the LLM’s generation process at query time. This is the essence of RAG, which the diagram effectively conveys: the LLM doesn’t work in isolation but is augmented by retrieval from a knowledge base.

**Note:** The diagram is conceptual; it abstracts some details (like how many chunks are retrieved, and it shows two document sources A and B in the answer). In practice, the answer format might be a narrative with or without explicit citations, but the system does keep track of source documents to provide if needed.

**9.3 Additional Diagram – Component Deployment (if needed)**

*(If the documentation format allows, we could include a simple deployment diagram showing how Docker containers are arranged: one for FastAPI, one for Postgres, etc. But since figure count/time is limited, we focus on the above two which capture core ideas.)*

The architecture and data flow diagrams together give a holistic picture:

* Figure 9.1 (Architecture) shows the system's static structure and integration points.
* Figure 9.2 (Data Flow) shows the runtime behavior for the primary use-case (QA query answering).

These visuals should help readers and stakeholders quickly grasp how the Local RAG system functions and how the components we described in text map into an orchestrated process delivering accurate, source-grounded answers.

In the next chapter, we will discuss potential future improvements to the system and additional features that could be added beyond the current implementation.

**10. Future Work and Potential Improvements**

While the project successfully met its objectives, there are several areas that offer room for enhancement and further development. This chapter outlines some key potential improvements and extensions that could be pursued in future work:

**10.1 Enhanced Document Processing and Coverage**

* **Support for More File Types:** Currently, we handle PDFs, text, and Word documents (the common ones for our use-case). Future work could integrate support for presentations (PPTX), spreadsheets (extracting text from cells), or even HTML/webpage ingestion. This would involve using libraries for those formats and might include special parsing (e.g., ignoring speaker notes in PPT or extracting readable text from HTML while stripping tags and scripts).
* **Optical Character Recognition (OCR):** As noted, our system doesn't handle scanned documents or images containing text. Integrating an OCR tool (like Tesseract or an OCR API) could allow ingesting scanned PDFs or images with text. This would broaden the applicability to archival documents or books that aren't digitized in text form. It's a non-trivial addition (OCR can introduce errors which propagate to embeddings), but with careful post-processing (e.g., manual corrections or confidence thresholds), it can be valuable.
* **Incremental / Real-time Document Indexing:** In the current setup, when a file is uploaded, the user must wait for it to be fully processed. A future improvement could be to handle large documents incrementally, so that initial parts become queryable even while later parts are still processing. This might involve a more complex pipeline (maybe splitting the document into parts and processing each concurrently). Additionally, a progress indicator on the front-end for indexing status would improve user experience for large uploads.

**10.2 Semantic Search Optimization and Scaling**

* **Metadata Filtering and Advanced Query:** We could extend the retrieval step to not only use pure vector similarity but also incorporate metadata filters. For instance, if documents have categories, dates, or authors, the user might query "What does *Document X* say about Y?" or "What are recent insights on Z (from documents after 2020)?". Implementing hybrid search (vector + structured filters) is possible with pgvector by combining vector similarity with traditional SQL conditions. This would allow more precise queries when needed.
* **Scaling to Large Corpora:** For significantly larger document collections (say tens of thousands of documents, millions of chunks), the performance might degrade or the memory usage of the index might grow. While Postgres with pgvector can handle moderately large sets, beyond a point, specialized vector databases (like Milvus, Pinecone) might be more efficient. Future work could evaluate when a switch is warranted and perhaps implement an abstraction to allow using an external vector DB if configured (similar to how we abstracted LLM providers). Alternatively, sharding the Postgres or using HNSW index (which pgvector supports) with tuned parameters might suffice. The key is ensuring our pipeline remains snappy as data scales, possibly by pre-filtering (e.g., first use keyword search to narrow docs, then vector search within them – an approach sometimes called "hybrid ranking").
* **Better Similarity Metrics:** We used cosine similarity via pgvector. There's potential to experiment with other techniques, such as **re-ranking**: after getting top-K by vector similarity, feed those chunks and the question into a smaller transformer or a BERT-based cross-encoder re-ranker to re-evaluate relevance. This could reorder results by a more context-aware measure. Cohere, for example, provides a re-rank API where you give a query and a list of texts and it scores them. If integrated, it might improve precision of which chunks we feed to the LLM (especially if K is large). This adds complexity and cost (another model call), so it would be optional if extreme accuracy is needed.

**10.3 Improved Answer Generation and Interaction**

* **Citation and Evidence Linking:** Currently, we provide source documents of the answer, but the answer text itself doesn't include footnote-style citations. A valuable improvement would be to train or prompt the LLM to output citations inline (e.g., "... catechins boost metabolism [1] ..."). We could then post-process those citation tags to link to the actual document or snippet. This would greatly enhance trust in the output, as users can see exactly which part of which document supports each statement. Achieving this might require a fine-tuned LLM or a careful prompt structure (some research has been done on citation in generation). Alternatively, one could do answer sentence-to-chunk attribution after generation: e.g., for each sentence the model produced, find the closest chunk from context and annotate it. This is an NLP task on its own but would be a cutting-edge feature.
* **Interactive Chat Memory:** Extending the system to support multi-turn conversations would be a natural next step. This means the user could ask follow-up questions like "What about black tea?" after asking about green tea, and the system should understand "black tea" in context of the prior conversation. Implementation wise, we can accumulate a history of Q&A and include a short summary of it in the prompt (ensuring we don't exceed token limits). Perhaps using the LLM to summarize the conversation so far into a compact form for context of the next question. Alternatively, one could maintain state on the backend (e.g., a conversation ID that ties to a list of relevant documents or earlier answers). This moves the project more into the realm of chatbots or conversational agents, which is popular and useful. It does require more prompt tokens though, so we'd have to balance the context length constraints.
* **Alternative Answer Modes:** Some use-cases might desire not just a direct answer, but other formats:
  + *Summaries:* e.g., "Summarize Document X". We could detect such queries and then instruct the LLM differently or possibly skip retrieval (if user explicitly wants whole doc summary, just pass the doc content if within limit, or do retrieval of all chunks).
  + *List or Bullet Answers:* If user asks for a list, we might set a format expectation in prompt ("Provide answer as a bullet list of points").
  + *Confidence and Uncertainty:* The system could quantify how confident it is. For instance, if the similarity scores of top chunks are low, we could add to the answer "The answer is not strongly supported by the documents." or have the model include a phrase like "I'm not entirely sure, but...". This would require either calibrating similarity to confidence or training a classifier on answer correctness (which is more research-oriented).
* **Multi-Language Support:** Currently, everything is in English (embedding model, docs assumed English, LLM outputs English). Future work could involve supporting other languages:
  + Cohere (and others) have multilingual embedding models which could be used.
  + If documents are in another language, we could either embed them with a multilingual model or translate them to English (via an API) then proceed, depending on needs.
  + The LLM side would need to handle the language (Google PaLM and DeepSeek might have multilingual capabilities). Alternatively, an intermediate step: if user asks a question in another language but docs are English, we translate question to English, do RAG, then translate answer back. This is a complex pipeline but could open up usage to non-English corpora or users.
  + A simpler initial step: ensure Unicode and diacritics are handled properly in parsing and embedding (which likely they are, but testing with e.g. a French PDF would be prudent).

**10.4 System Integration and Deployment**

* **Mobile and Desktop Applications:** While we built a web interface, packaging the solution into a desktop app (using Electron, for example) or a mobile app could be useful for offline or on-premise usage. Imagine a researcher who has a personal knowledge base on their computer; a desktop app with an embedded local LLM (like DeepSeek 7B) and the RAG logic could run without internet. This requires creating an Electron UI that wraps our front-end, or writing a simple Python GUI (PyQt etc.) interacting with the backend logic. Similarly, a mobile app could allow taking a photo of a document (OCR it) and then asking questions—combining a lot of the improvements suggested.
* **Container Orchestration for Microservices:** If this project were to grow, splitting services might be prudent (embedding service as separate microservice, etc.). Future deployment might use Kubernetes to scale components independently. For instance, if the usage pattern shows the LLM service is the bottleneck, one might want to scale out multiple LLM generator instances behind a queue. Implementing a task queue (Celery, RabbitMQ etc.) for queries could allow smoothing out bursts of requests, though at some latency cost. This moves beyond the scope of a single-machine deployment to a robust cloud deployment scenario, something to consider for enterprise adoption of such a system.
* **Logging and Monitoring:** Future versions should include better logging (structured logs of each query, which chunks were used, what answer was given). This not only helps in debugging and improving the system (by analyzing where it might have given wrong answers) but is also important for trust—keeping an audit trail of what sources were used. We did include source info, but robust monitoring might involve tracking embedding service latency, LLM latency, etc. Tools like Prometheus for metrics or Sentry for error tracking could be integrated. This is not a user-facing feature, but important for maintenance if the system is deployed long-term.

**10.5 Research Extensions**

* **Learning to Retrieve:** An advanced line of future work is to incorporate learning-based retrievers. Instead of using static embeddings, one could train a model (like DPR – Dense Passage Retriever, or use models like sentence-transformers fine-tuned on Q&A pairs) to get potentially better retrieval specific to our domain. This would require creating (or obtaining) a dataset of questions and relevant document segments to fine-tune on. It's a complex improvement and might not be needed for decent performance, but could be a research project offshoot to see if it beats the generic embedding model in our context.
* **Feedback Loop:** We could allow users to give feedback on answers (e.g., a thumbs up/down). This feedback could be used to improve the system over time. For instance, if user says an answer was not correct or not useful, we might analyze whether the right context was retrieved or if the LLM misunderstood. Over time, this could lead to either adjusting the pipeline (e.g., maybe always retrieve more context for certain question types) or train a model on the feedback (reinforcement learning or at least a classifier to detect low-quality answers before showing them). This enters the realm of Human-in-the-Loop training, which is quite advanced but potentially powerful—think of it as making our QA assistant learn from its mistakes.
* **Privacy-preserving enhancements:** If this system were used with sensitive data, future work might involve ensuring that the LLM (if cloud-based) does not get to see data it shouldn't. This could involve on-the-fly anonymization of certain terms in the prompt (like replace names with placeholders) if using third-party LLMs, or focusing on local models for certain categories of documents. Also, encryption of the vector store (use Postgres encryption or encrypt documents at rest) could be considered.

Each of these potential improvements could be a project on its own. Depending on future needs or interests, one might prioritize some over others. For instance, in an academic context, the research extensions (like improving retrieval or integrating feedback) might be most interesting. In a product context, user experience improvements (multi-turn chat, multi-language) could be more immediately valuable.

In summary, the Local RAG system we've built is a solid foundation, and these future directions show that there's plenty of scope to extend its capabilities. By addressing these, the system could become more versatile, accurate, and user-friendly, potentially evolving from a prototype into a production-grade solution or an advanced research platform for retrieval-augmented generation techniques.

**11. Appendices**

In this final section, we provide supplementary material that supports the main content of the document. This includes a glossary of technical terms, a list of figures and tables used in the documentation, and a bibliography of references for further reading and verification of information.

**Appendix A: Glossary of Terms**

* **RAG (Retrieval-Augmented Generation):** An approach in AI that combines information retrieval with text generation. The system first fetches relevant data (e.g., documents) for a query and then uses a language model to generate an answer grounded in that data.
* **Embedding (Vector Embedding):** A numerical representation of data (such as text) in a high-dimensional space. Similar pieces of text have vectors that are close to each other in this space. Used for measuring semantic similarity between texts.
* **Vector Database:** A database optimized for storing and querying vectors (embeddings). It enables similarity search – finding the nearest vectors to a given query vector. Pgvector is an extension that adds this capability to PostgreSQL.
* **Similarity Search (Semantic Search):** The process of finding data that is semantically related to a query by comparing embedding vectors. Unlike keyword search, it can match concepts even if exact words differ.
* **Transformer:** A type of neural network architecture known for its self-attention mechanism. Transformers power modern language models (e.g., BERT, GPT) and can generate context-aware embeddings and text.
* **LLM (Large Language Model):** A very large Transformer-based model trained on massive text corpora. It can generate human-like text. Examples include OpenAI’s GPT-3/GPT-4, Google’s PaLM, and open models like DeepSeek LLM.
* **Cohere:** An AI platform offering NLP models via API. In our project, Cohere’s embedding API converts text into vectors capturing semantic meaning.
* **Google PaLM API:** Google’s service for their Pathways Language Model, accessible to developers. We use it as one option to generate answers from provided context.
* **DeepSeek LLM:** An open-source large language model (available in 7B and 67B parameter versions) noted for high performance in reasoning tasks. We integrate it as a local model for answering queries without external API calls.
* **FastAPI:** A modern Python web framework for building APIs quickly and efficiently. It provides features like automatic documentation and async support, used for our backend server.
* **pgvector:** A PostgreSQL extension that introduces a vector data type and similarity search operators. It allows Postgres to function as a vector database, used to store and query text embeddings in our system.
* **IVFFLAT (Index):** An algorithm for approximate nearest neighbor search. In pgvector, an IVFFLAT index partitions vectors into clusters for faster search at slight cost to exact accuracy. We use it to speed up similarity queries.
* **OCR (Optical Character Recognition):** Technology to convert images of text (scans, photos) into actual text characters. Not implemented in our project, but mentioned as a future improvement for ingesting scanned documents.
* **Cosine Similarity:** A metric that measures the cosine of the angle between two vectors. Ranges from -1 to 1, where 1 means vectors are identical in direction (i.e., very similar in meaning). Often used in embedding comparisons (pgvector uses an operator <-> for distance based on cosine or Euclidean).
* **Pydantic:** A Python library for data validation. Used by FastAPI to define request and response schemas. Ensures input data types are correct (e.g., that a query JSON has a string field "question").
* **Docker:** A containerization platform that packages software and its dependencies into containers. We used Docker for deploying our backend, database, and optionally the frontend, ensuring consistency across environments.
* **Swagger UI / OpenAPI Docs:** Automatically generated web documentation for APIs. FastAPI generates this (the interactive docs interface) so that during development or use one can test endpoints easily.

**Appendix B: List of Figures and Tables**

* *Figure 9.1: System Architecture Overview.* – A diagram illustrating the components of the system and how they interact (user interface, backend services, database, external APIs) (Section 9.1).
* *Figure 9.2: Data Flow in RAG Pipeline.* – A sequence diagram showing how a user’s query is processed: from embedding to retrieval to generation of the answer (Section 9.2).

*(Note: There are no additional tables in the document; all information has been conveyed in text or figure form.)*

**Appendix C: Bibliography (References)**

The following references include academic papers, technical blogs, and documentation that were cited in the text and that informed the development of this project:

1. **Lewis et al., 2020 – Retrieval-Augmented Generation (RAG) Paper:** Mike Lewis et al., *"Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks,"* NeurIPS 2020. This paper introduced the RAG concept, combining retrieval with generation. It demonstrated that augmenting a language model with a retrieved knowledge source can improve accuracy on knowledge-intensive tasks.
2. **NVIDIA Technical Blog (2023):** Rick Merritt, *"What Is Retrieval-Augmented Generation, aka RAG?"*, NVIDIA Blog, Jan 31, 2025. Provides an overview of RAG with analogies and notes the origin of the term by Patrick Lewis. It emphasizes how RAG enhances accuracy and reliability of LLMs by grounding them in external data.
3. **Cohere Documentation – RAG and Embeddings:** Cohere, *"Retrieval Augmented Generation (RAG)"* [Documentation] and Cohere API Reference for *Embed*. The docs explain RAG as using external data to improve model responses, reducing hallucinations. The embedding reference defines what an embedding is (list of floats capturing semantic info) and its use in semantic search.
4. **Medium Article on Semantic Search:** Sudhir Yelikar, *"Understanding similarity or semantic search and vector databases,"* Medium, May 24, 2023. Explains how vector embeddings represent data and how similarity is calculated (with similar data having closer vectors). It outlines the process of creating embeddings for queries and doing nearest neighbor search.
5. **PostgreSQL + pgvector Introduction:** Sabrina Aquino, Qdrant Tech Blog, *"An Introduction to Vector Databases,"* Oct 09, 2024. A quote from this states, "A Vector Database is a specialized system designed to efficiently handle high-dimensional vector data... enabling similarity searches that traditional databases cannot easily perform." This justified our use of pgvector in Postgres.
6. **Pinecone Vector DB Explanation:** Roie Schwaber-Cohen, Pinecone Blog, *"What is a Vector Database & How Does it Work?"* May 3, 2023. Provides a similar definition that a vector DB indexes and stores embeddings for fast similarity search. Underlined the need for such specialized storage when dealing with AI embeddings.
7. **DeepSeek LLM GitHub README:** *DeepSeek LLM: Let there be answers,* deepseek-ai on GitHub (2024). Describes DeepSeek as an advanced open-source model (67B parameters, trained on 2T tokens) and notes its performance surpassing Llama2 70B in certain aspects. This informed our choice to experiment with it as a local model option.
8. **DataCamp – Transformers and Self-Attention:** *"Attention Mechanism in LLMs: An Intuitive Explanation,"* DataCamp, Apr 26, 2024. Clarifies how self-attention in Transformers works, allowing models to weigh the importance of words relative to others, capturing contextual relationships. This is foundational to understanding why Transformers (like our LLMs) can effectively use inserted context (they attend to provided documents when generating answers).
9. **Cohere blog on RAG techniques:** Cohere, *"RAG enables a model to access and utilize supplementary information..."* (from their docs or blog). This provided insight into RAG’s benefits such as improved accuracy, which we cited to emphasize motivation for our approach.
10. **OpenAI / Azure references (if any)** – *[No direct citations]*, but knowledge of environment variables like OPENAI\_API\_KEY and usage came from OpenAI’s own documentation and widely-known usage patterns.
11. **Project Mini-RAG original repository & course by Abu Bakr Soliman:** The YouTube videos and GitHub (bakrianoo/mini-rag) served as a base guidance (though not directly cited in content beyond context). For example, the branching and steps we followed were inspired by that educational series.

All web citations with the format 【source†Lx-Ly】 refer to specific lines in the referenced documents for precise context. These sources provided both conceptual background and practical guidance during implementation, and the bibliography ensures proper credit and allows readers to find more detailed information if desired.