# Developing Retrieval Augmented Generation (RAG) based LLM Systems from PDFs: An Experience Report

**Ayman Asad Khan**
Tampere University
ayman.khan@tuni.fi

**Md Toufique Hasan**
Tampere University
mdtoufique.hasan@tuni.fi

**Kai Kristian Kemell**
Tampere University
kai-kristian.kemell@tuni.fi

**Jussi Rasku**
Tampere University
jussi.rasku@tuni.fi

**Pekka Abrahamsson**
Tampere University
pekka.abrahamsson@tuni.fi

**Abstract.** This paper presents an experience report on the development of Retrieval Augmented Generation (RAG) systems using PDF documents as the primary data source. The RAG architecture combines generative capabilities of Large Language Models (LLMs) with the precision of information retrieval. This approach has the potential to redefine how we interact with and augment both structured and unstructured knowledge in generative models to enhance transparency, accuracy and contextuality of responses. The paper details the end-to-end pipeline, from data collection, preprocessing, to retrieval indexing and response generation, highlighting technical challenges and practical solutions. We aim to offer insights to researchers and practitioners developing similar systems using two distinct approaches: *OpenAI's Assistant API with GPT Series* and *Llama's open-source models*. The practical implications of this research lie in enhancing the reliability of generative AI systems in various sectors where domain specific knowledge and real time information retrieval is important. The Python code used in this work is also available at: GitHub.

**Keywords:** Retrieval Augmented Generation (RAG), Large Language Models (LLMs), Generative AI in Software Development, Transparent AI.

## 1 Introduction

Large language models (LLMs) excel at generating human like responses, but base AI models can't keep up with the constantly evolving information within dynamic sectors. They rely on static training data, leading to outdated or incomplete answers. Thus they often lack transparency and accuracy in high stakes

decision making. Retrieval Augmented Generation (RAG) presents a powerful solution to this problem. RAG systems pull in information from external data sources, like PDFs, databases, or websites, grounding the generated content in accurate and current data making it ideal for knowledge intensive tasks.

In this report, we document our experience as a step-by-step guide to build RAG systems that integrates PDF documents as the primary knowledge base. We discuss the design choice, development of system, and evaluation of the guide, providing insights into the technical challenges encountered and the practical solutions applied. We detail our experience using both proprietary tools (OpenAI) and open-source alternatives (Llama) with data security, offering guidance on choosing the right strategy. Our insights are designed to help practitioners and researchers optimize RAG models for precision, accuracy and transparency that best suites their use case.

## 2 Background

This section presents the theoretical background of this study. Traditional generative models, such as GPT, BERT, or T5 are trained on massive datasets but have a fixed internal knowledge cut off based on their training data. They can only generate **black box** answers based on what they **know**, and this limitation is notable in fields where information changes rapidly and better explainability and traceability of responses is required, such as healthcare, legal analysis, customer service, or technical support.

### 2.1 What is RAG?

The concept of Retrieval Augmented Generation (RAG) models is built on integrating two core components of NLP: Information Retrieval (IR) and Natural Language Generation (NLG). The RAG framework, first introduced by Lewis et al.[5] combines dense retrieval methods with large scale generative models to produce responses that are both contextually relevant and factually accurate. By explicitly retrieving relevant passages from a large corpus and augmenting this information in the generation process, RAG models enhance the factual grounding of their outputs from the up-to-date knowledge.

A generic workflow of Retrieval Augmented Generation (RAG) system, showcasing how it fundamentally enhances the capabilities of Large Language Models (LLMs) by grounding their outputs in real-time, relevant information is illustrated in the Fig[1]. Unlike static models which generate responses based only on closed-world knowledge, the RAG process is structured into the following key steps:

1. **Data Collection:**
   The workflow begins with the acquisition of relevant, domain specific textual data from various external sources, such as PDFs, structured documents, or text files. These documents represent raw data important for building a tailored knowledge base that the system will query during the retrieval process

enhancing the model's ability to respond.

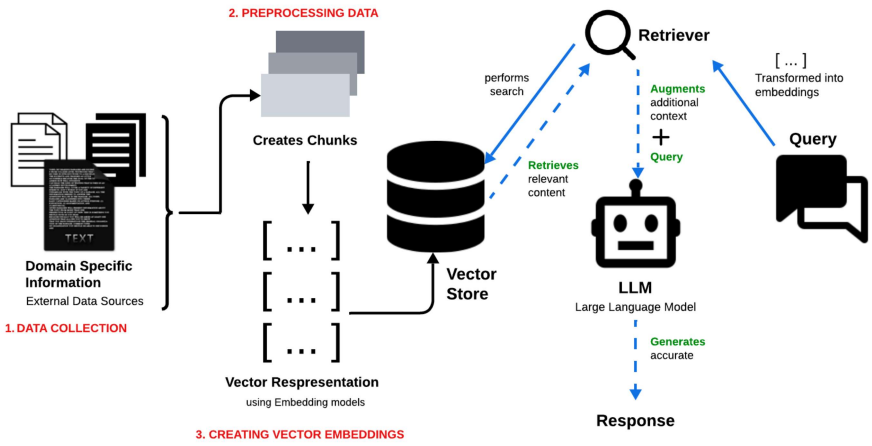


Fig. 1: Architecture of Retrieval Augmented Generation(RAG) system.

2. **Data Preprocessing:**
   The collected data is then preprocessed to create manageable and meaningful chunks. Preprocessing involves cleaning the text (e.g., removing noise, formatting), normalizing it, and segmenting it into smaller units, such as *tokens* (e.g., words or group of words), that can be easily indexed and retrieved later. This segmentation is necessary to ensure that the retrieval process is accurate and efficient.

3. **Creating Vector Embeddings:**
   After preprocessing, the chunks of data are transformed into *vector representations* using embedding models (e.g., BERT, Sentence Transformers). These vector embeddings capture the semantic meaning of the text, allowing the system to perform similarity searches. The vector representations are stored in a **Vector Store**, an indexed database optimized for fast retrieval based on similarity measures.

4. **Retrieval of Relevant Content:**
   When a **Query** is input into the system, it is first transformed into a *vector embedding*, similar to the documents in the vector store. The **Retriever** component then performs a search within the vector store to identify and retrieve the most relevant chunks of information related to the query. This retrieval process ensures that the system uses the most pertinent and up-to-

date information to respond to the query.

5. **Augmentation of Context:**
   By merging two knowledge streams - the fixed, general knowledge embedded in the LLM and the flexible, domain-specific information augmented on demand as an additional layer of context, aligns the Large Language Model (LLM) with both established and emerging information.

6. **Generation of Response by LLM:**
   The *context-infused prompt*, consisting of the original user query combined with the retrieved relevant content is provided to a Large Language Model (LLM) like GPT, T5 or Llama. The LLM then processes this augmented input to generate a coherent response not only fluent but factually grounded.

7. **Final Output:**
   By moving beyond the opaque outputs of traditional models, the final output of RAG systems offer several advantages: they minimize the risk of generating hallucinations or outdated information, enhance interpretability by clearly linking outputs to real-world sources, enriched with relevant and accurate responses.

The RAG model framework introduces a paradigm shift in Generative AI by creating **glass-box** models. It greatly enhanced the ability of generative models to provide accurate information, especially in knowledge-intensive domains. This integration has become the backbone of many advanced NLP applications, such as chatbots, virtual assistants, and automated customer service systems.[5]

### 2.2 When to Use RAG: Considerations for Practitioners

Choosing between fine-tuning, using Retrieval Augmented Generation (RAG), or base models can be a challenging decision for practitioners. Each approach offers distinct advantages depending on the context and constraints of the use case. This section aims to outline the scenarios in which each method is most effective, providing a decision framework to guide practitioners in selecting the appropriate strategy.

**2.2.1 Fine-Tuning: Domain Expertise and Customization** Fine-tuning involves training an existing large language model (LLM) on a smaller, specialized dataset to refine its knowledge for a particular domain or task. This method excels in scenarios where accuracy, tone consistency, and deep understanding of niche contexts are essential. For instance, fine-tuning has been shown to improve a model's performance in specialized content generation, such as technical writing, customer support, and internal knowledge systems.

**Advantages:** Fine-tuning embeds domain specific knowledge directly into the model, reducing the dependency on external data sources. It is particularly

effective when dealing with stable data or when the model needs to adhere to a specific tone and style.

**Drawbacks:** Fine-tuning is computationally expensive and often requires substantial resources for initial training. Additionally, it risks overfitting if the dataset is too narrow, making the model less generalizable.

**Use Case Examples:**

- **Medical Diagnosis**: A fine-tuned model on medical datasets becomes highly specialized in understanding and generating medical advice based on specific terminologies and contexts.
- **Customer Support**: For a software company, fine-tuning on company-specific troubleshooting protocols ensures high-accuracy and consistent responses tailored to user queries.

**2.2.2 RAG: Dynamic Information and Large Knowledge Bases** Retrieval-Augmented Generation (RAG) combines LLMs with a retrieval mechanism that allows the model to access external data sources in real-time, making it suitable for scenarios requiring up-to-date or frequently changing information. RAG systems are valuable for handling vast knowledge bases, where embedding all the information directly into the model would be impractical or impossible.

**Advantages:** RAG is ideal for applications that require access to dynamic information, ensuring responses are grounded in real-time data and minimizing hallucinations. It also provides transparency, as the source of the retrieved information can be linked directly.

**Drawbacks:** RAG requires complex infrastructure, including vector databases and effective retrieval pipelines, and can be resource-intensive during inference.

**Use Case Examples:**

- **Financial Advisor Chatbot**: Using RAG, a chatbot can pull the latest market trends and customer-specific portfolio data to offer personalized investment advice.
- **Legal Document Analysis**: RAG can retrieve relevant case laws and statutes from a constantly updated database, making it suitable for legal applications where accuracy and up-to-date information are critical.

**2.2.3 When to Use Base Models** Using base models(without fine-tuning or RAG) is appropriate when the task requires broad generalization, low-cost deployment, or rapid prototyping. Base models can handle simple use cases like generic customer support or basic question answering, where specialized or dynamic information is not required.

**Advantages:** No additional training is required, making it easy to deploy and maintain. It is best for general purpose tasks or when exploring potential applications without high upfront costs.

**Drawbacks:** Limited performance on domain specific queries or tasks that need high levels of customization.

Table 1: Decision Framework for Choosing Between Fine-Tuning, RAG, and Base Models

| **Factors** | **Fine-Tuning** | **RAG** | **Base Models** |
|---|---|---|---|
| **Nature of the Task** | Highly specialized tasks, domain specific language | Dynamic tasks needing real time information retrieval | General tasks, prototyping, broad applicability |
| **Data Requirements** | Static or proprietary data that rarely changes | Access to up-to-date or external large knowledge bases | Does not require specialized or up-to-date information |
| **Resource Constraints** | High computational resources needed for training | Higher inference cost and infrastructure complexity | Low resource demand, quick to deploy |
| **Performance Goals** | Maximizing precision and adaptability to specific language | Providing accurate, context-aware responses from dynamic sources | Optimizing speed and cost efficiency over precision |

To conclude, the decision framework outlined in Table[1] offers practitioners a guide to selecting the most suitable method based on their project's specific needs. Fine-Tuning is the best option for specialized, high-precision tasks with stable data; RAG should be used when access to dynamic, large-scale data is necessary; and Base Models are well-suited for general-purpose use with low resource requirements.

### 2.3 Understanding the Role of PDFs in RAG

PDFs are paramount for RAG applications because they are widely used for distributing high-value content like research papers, legal documents, technical manuals, and financial reports, all of which contain dense, detailed information essential for training RAG models. PDFs come in various forms, allowing access to a wide range of data types—from scientific data and technical diagrams to legal terms and financial figures. This diversity makes PDFs an invaluable resource for extracting rich, contextually relevant information. Additionally, the consistent formatting of PDFs ensures accurate text extraction and context preservation, which is fundamental for generating precise responses. PDFs also include metadata (like author, keywords, and creation date) and annotations (such as

highlights and comments) that provide extra context, helping RAG models prioritize sections and better understand document structure, ultimately enhancing retrieval and generation accuracy.

**2.3.1 Challenges of Working with PDFs** In RAG applications, accurate text extraction from PDFs is essential for effective retrieval and generation. However, PDFs often feature complex layouts—such as multiple columns, headers, footers, and embedded images—that complicate the extraction process. These complexities challenge RAG systems, which rely on clean, structured text for high-quality retrieval. Text extraction accuracy from PDFs decreases dramatically in documents with intricate layouts, such as multi-column formats or those with numerous figures and tables. This decline necessitates advanced extraction techniques and machine learning models tailored to diverse document structures.

Moreover, the lack of standardization in PDF creation, including different encoding methods and embedded fonts, can result in inconsistent or garbled text, further complicating extraction and degrading RAG model performance. Additionally, many PDFs are scanned documents, especially in fields like law and academia, requiring Optical Character Recognition (OCR) to convert images to text. OCR can introduce errors, particularly with low-quality scans or handwritten text, leading to inaccuracies that are problematic in RAG applications, where precise input is essential for generating relevant responses. PDFs may also contain non-textual elements like charts, tables, and images, disrupting the linear text flow required by most RAG models. Handling these elements requires specialized tools and preprocessing to ensure the extracted data is coherent and useful for RAG tasks.

**2.3.2 Key Considerations for PDF Processing in RAG Application Development** Processing PDFs for Retrieval Augmented Generation (RAG) applications requires careful handling to ensure high-quality text extraction, effective retrieval, and accurate generation. Below are key considerations specifically tailored for PDF processing in RAG development.

***1. Accurate Text Extraction:***
Since PDFs can have complex formatting, it is essential to use reliable tools and methods to convert the PDF content into usable text for further processing.

- **Appropriate Tool for Extraction:** There are tools and libraries for extracting text from PDFs for most popular programming languages (i.e: `pdfplumber` or `PyMuPDF (fitz)` for Python). These libraries handle most common PDF structures and formats, preserving the text's layout and structure as much as possible.
- **Verify and Clean Extracted Text:** After extracting text, always verify it for completeness and correctness. This step is essential for catching any extraction errors or artifacts from formatting.

***2. Effective Chunking for Retrieval:***
PDF documents often contain large blocks of text, which can be challenging for retrieval models to handle effectively. Chunking the text into smaller, contextually coherent pieces can improve retrieval performance.

- **Semantic Chunking:** Instead of splitting text arbitrarily, use semantic chunking based on logical divisions within the text, such as paragraphs or sections. This ensures that each chunk retains its context, which is important for both retrieval accuracy and relevance.
- **Dynamic Chunk Sizing:** Adjust the chunk size according to the content type and the model's input limitations. For example, scientific documents might be chunked by sections, while other types of documents could use paragraphs as the primary chunking unit.

***3. Preprocessing and Cleaning:***
Preprocessing the extracted text is key for removing noise that could affect the performance of both retrieval and generative models. Proper cleaning ensures the text is consistent, relevant, and ready for further processing.

- **Remove Irrelevant Content:** Use regular expressions or NLP-based rules to clean up non-relevant content like headers, footers, page numbers, and any repeating text that doesn't contribute to the document's meaning.
- **Normalize Text:** Standardize the text format by converting it to lowercase, removing special characters, and trimming excessive whitespace. This normalization helps create consistent input for the retrieval models.

***4. Utilizing PDF Metadata and Annotations:***
PDFs often contain metadata (such as the author, title, and creation date) and annotations that provide additional context, which can be valuable for retrieval tasks in RAG applications.

- **Extract Metadata:** You can use tools specific to programming languages like `PyMuPDF` or `pdfminer.six` for Python to extract embedded metadata. This metadata can be used as features in retrieval models, adding an extra layer of context for more precise search results.
- **Utilize Annotations:** Extract and analyze annotations or comments within PDFs to understand important or highlighted sections. This can help prioritize content in the retrieval process.

***5. Error Handling and Reliability:***
Reliability in processing PDFs is essential for maintaining the stability and reliability of RAG applications. Implementing proper error handling and logging helps manage unexpected issues and ensures smooth operation.

- **Implement Error Handling:** Use try-except blocks to manage potential errors during PDF processing. This ensures the application continues running smoothly and logs any issues for later analysis.

– **Use Logging for Monitoring:** Implement logging to capture detailed information about the PDF processing steps, including successes, failures, and any anomalies. This is important for debugging and optimizing the application over time.

By following these key considerations and best practices, we can effectively process PDFs for RAG applications, ensuring high-quality text extraction, retrieval, and generation. This approach ensures that your RAG models are strong, efficient, and capable of delivering meaningful insights from complex PDF documents.

## 3 Study Design

This section presents the methodology for building a Retrieval Augmented Generation (RAG) system that integrates PDF documents as a primary knowledge source. This system combines the retrieval capabilities of information retrieval (IR) techniques with the generative strengths of Large Language Models (LLMs) to produce factually accurate and contextually relevant responses, grounded in domain-specific documents.

The goal is to design and implement a RAG system that addresses the limitations of traditional LLMs, which rely solely on static, pre-trained knowledge. By incorporating real-time retrieval from domain-specific PDFs, the system aims to deliver responses that are not only contextually appropriate but also up-to-date and factually reliable.

The system begins with the collection of relevant PDFs, including research papers, legal documents, and technical manuals, forming a specialized knowledge base. Using tools and libraries, the text is extracted, cleaned, and preprocessed to remove irrelevant elements such as headers and footers. The cleaned text is then segmented into manageable chunks, ensuring efficient retrieval. These text segments are converted into vector embeddings using transformer-based models like BERT or Sentence Transformers, which capture the semantic meaning of the text. The embeddings are stored in a vector database optimized for fast similarity-based retrieval.

The RAG system architecture consists of two key components: a retriever, which converts user queries into vector embeddings to search the vector database, and a generator, which synthesizes the retrieved content into a coherent, factual response. Two types of models are considered: OpenAI's GPT models, accessed through the Assistant API for ease of integration, and the open-source Llama model, which offers greater customization for domain-specific tasks.

In developing the system, several challenges are addressed, such as managing complex PDF layouts (e.g., multi-column formats, embedded images) and maintaining retrieval efficiency as the knowledge base grows. These challenges were highlighted during a preliminary evaluation process, where participants pointed out the difficulty of handling documents with irregular structures. Feedback from the evaluation also emphasized the need for improvements in text extraction and chunking to ensure coherent retrieval.

The design also incorporates the feedback from a diverse group of participants during a workshop session, which focused on the practical aspects of implementing RAG systems. Their input highlighted the effectiveness of the system's real-time retrieval capabilities, particularly in knowledge-intensive domains, and underscored the importance of refining the integration between retrieval and generation to enhance the transparency and reliability of the system's outputs. This design sets the foundation for a RAG system capable of addressing the needs of domains requiring precise, up-to-date information.

# 4 Results: Step-by-Step Guide to RAG

## 4.1 Setting Up the Environment

This section walks you through the steps required to set up a development environment for Retrieval Augmented Generation (RAG) on your local machine. We will cover the installation of Python, setting up a virtual environment and configuring an IDE (VSCode).

**4.1.1 Installing Python** If Python is not already installed on your machine, follow the steps below:

1. **Download and Install Python**
   - Navigate to the official Python website: https://www.python.org/downloads/
   - Download the latest version of Python for your operating system (Windows, macOS, or Linux).
   - During installation, ensure that you select the option *Add Python to PATH*. This is important to run Python from the terminal or command line.
   - For Windows users, you can also:
     - Click on *Customize Installation*.
     - Select *Add Python to environment variables*.
     - Click *Install Now*.

2. **Verify the Installation**
   - Open the terminal (Command Prompt on Windows, Terminal on macOS/Linux).
   - Run the following command to verify that Python is installed correctly: `python --version`
   - If Python is installed correctly, you should see output similar to `Python 3.x.x`.

**4.1.2 Setting Up an IDE** After installing Python, the next step is to set up an Integrated Development Environment (IDE) to write and execute your Python code. We recommend Visual Studio Code (VSCode), however you are free to choose editor of your own choice. Below are the setup instructions for VSCode.

1. **Download and Install VSCode**
   - Visit the official VSCode website: https://code.visualstudio.com/.
   - Select your operating system (Windows, macOS, or Linux) and follow the instructions for installation.

2. **Install the Python Extension in VSCode**
   - Open VSCode.
   - Click on the *Extensions* tab on the left-hand side (it looks like a square with four pieces).
   - In the Extensions Marketplace, search for *Python*.
   - Install the Python extension by Microsoft. This will allow VSCode to support Python code.

**4.1.3 Setting Up a Virtual Environment** A virtual environment allows you to install libraries and dependencies specific to your project without affecting other projects on your machine.

1. **Open the Terminal in VSCode**
   - Press `Ctrl + ‘` (or `Cmd + ‘` on Mac) to open the terminal in VSCode.
   - Alternatively, navigate to *View* - *Terminal* in the menu.
   - In the terminal, use the `mkdir` command to create a new folder for your project. For example, to create a folder named `my-new-project`, type:
     ```
     mkdir my-new-project
     ```
   - Use the `cd` command to change directories and navigate to the folder where your project is located. For example:
     ```
     cd path/to/your/project/folder/my-new-project
     ```
2. **Create a Virtual Environment**
   - For Windows, run the following commands:
     ```
     python -m venv my_rag_env
     my_rag_env\Scripts\activate
     ```
   - For Mac/Linux, run the following commands:
     ```
     python3 -m venv my_rag_env
     source my_rag_env/bin/activate
     ```
3. **Configure VSCode to Use the Virtual Environment**
   - Open the Command Palette by pressing `Ctrl + Shift + P` (or `Cmd + Shift + P` on Mac).
   - Type *Python: Select Interpreter* in the Command Palette.
   - Select your virtual environment, `my_rag_env`, from the list.

With your virtual environment now configured, you are ready to install project specific dependencies and manage Python packages independently for each approach. This setup allows you to create separate virtual environments for the two approaches outlined in Sections[4.2.1][4.2.2]. By isolating your dependencies, you can ensure that the OpenAI Assistant API-based[4.2.1] and Llama-based [4.2.2] Retrieval Augmented Generation (RAG) systems are developed and managed in their respective environments without conflicts or dependency issues. This practice also helps maintain cleaner, more manageable development workflows for both models, ensuring that each approach functions optimally with its specific requirements.

### 4.2 Two Approaches to RAG: Proprietary and Open source

This section introduces a structured guide for developing Retrieval Augmented Generation (RAG) systems, focusing on two distinct approaches: *using OpenAI's Assistant API (GPT Series)* and an *open-source Large Language Model (LLM) Llama* and thus divided into two subsections[4.2.1][4.2.2]. The objective is to equip developers with the knowledge and practical steps necessary to implement RAG systems effectively, while highlighting common mistakes and best practices at each stage of the process. Each subsection is designed to provide practical insights into setup, development, integration, customization and optimization to generate well-grounded and aligned outputs.

In addition to the two primary approaches discussed in this guide there are several alternative frameworks and methodologies for developing Retrieval Augmented Generation (RAG) systems. Each of these options such as Cohere, AI21's Jurassic-2, Google's PaLM, and Meta's OPT have their merits and trade-offs in terms of deployment flexibility, cost, ease of use, and performance.

We have selected **OpenAI's Assistant API (GPT Series)** and **Llama** for this guide based on their wide adoption, proven capabilities, and distinct strengths in developing RAG systems. As highlighted in comparison Table[2] *OpenAI's Assistant API* provides a simple and developer-friendly black-box, allowing quick integration and deployment without the need for extensive model management or infrastructure setup with high quality outputs. In contrast, as an open-source model, Llama allows developers to have full control over the model's architecture, training data, and fine-tuning process, allowing for precise customization to suit specific requirements such as demand control, flexibility, and cost-efficiency. This combination makes these two options highly valuable for diverse RAG system development needs.

Table 2: Comparison of RAG Approaches: OpenAI vs. Llama

| **Feature** | **OpenAI's Assistant API (GPT Series)** | **Llama (Open-Source LLM Model)** |
|---|---|---|
| **Ease of Use** | High. Simple API calls with no model management | Moderate. Requires setup and model management |
| **Customization** | Limited to prompt engineering and few-shot learning | High. Full access to model fine-tuning and adaptation |
| **Cost** | Pay-per-use pricing model | Upfront infrastructure costs; no API fees |
| **Deployment Flexibility** | Cloud-based; depends on OpenAI's infrastructure | Highly flexible; can be deployed locally or in any cloud environment |
| **Performance** | Excellent for a wide range of general NLP tasks | Excellent, particularly when fine-tuned for specific domains |
| **Security and Data Privacy** | Data is processed on OpenAI servers; privacy concerns may arise | Full control over data and model; suitable for sensitive applications |
| **Support and Maintenance** | Strong support, documentation, and updates from OpenAI | Community-driven; updates and support depend on community efforts |
| **Scalability** | Scalable through OpenAI's cloud infrastructure | Scalable depending on infrastructure setup |
| **Control Over Updates** | Limited; depends on OpenAI's release cycle | Full control; users can decide when and how to update or modify the model |

**4.2.1 Using OpenAI's Assistant API : GPT Series** While the OpenAI Completion API is effective for simple text generation tasks, the Assistant API is a superior choice for developing RAG systems. The Assistant API supports **multi-modal operations** (such as text, images, audio, and video inputs) by combining text generation with file searches, code execution, and API calls. For a RAG system, this means an assistant can retrieve documents, generate vector embeddings, search for relevant content, augment user queries with additional context, and generate responses—all in a seamless, integrated workflow. It includes memory management across sessions, so the assistant *remembers* past queries, retrieved documents, or instructions. Assistants can be configured with specialized instructions, behaviors, parameters other than custom tools that makes this API far more powerful for developing RAG systems.

This subsection provides a step-by-step guide and code snippets to utilize the OpenAI's **File Search tool** within the Assistant API, as illustrated in