

See discussions, stats, and author profiles for this publication at: <https://www.researchgate.net/publication/385108254>

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

Preprint · October 2024

DOI: 10.48550/arXiv.2410.15944

CITATION

1

READS

1,418

5 authors, including:



Md Toufique Hasan

Tampere University

4 PUBLICATIONS 15 CITATIONS

[SEE PROFILE](#)



Pekka Abrahamsson

Tampere University

363 PUBLICATIONS 12,157 CITATIONS

[SEE PROFILE](#)

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 They rely on static training data, leading to outdated or incomplete answers.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.

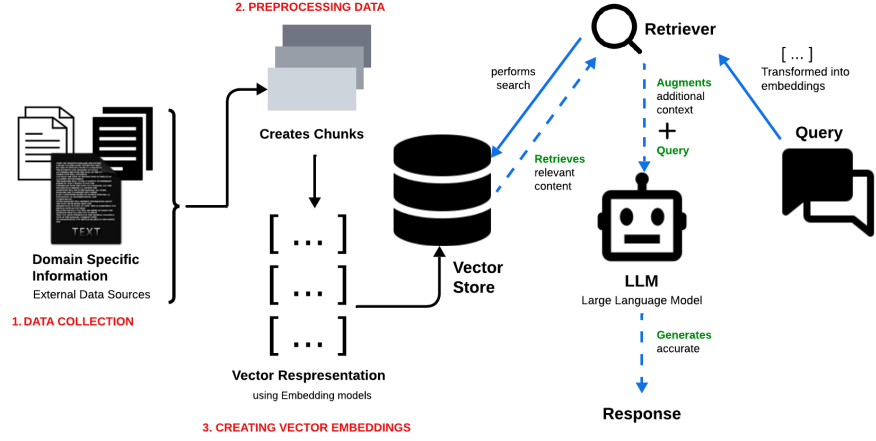


Fig. 1: Architecture of Retrieval Augmented Generation(RAG) system.

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 lower-case, 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

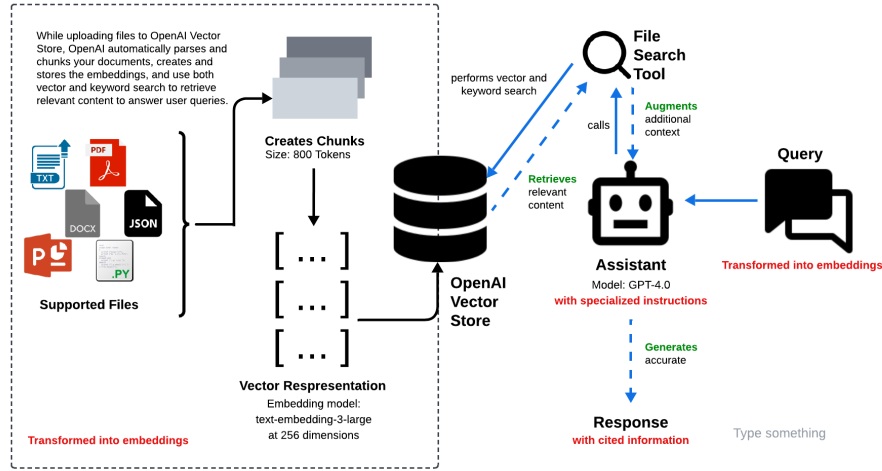


Fig. 2: Open AI's Assistant API Workflow

Fig[2] to implement RAG. The diagram shows how after the domain specific data ingestion of supported files (such as PDFs, DOCX, JSON, etc.), the data preprocessing[2] and vectorization[3] is handled by Assistant API. These vectors are stored in OpenAI **Vector Store**, which the File Search tool can query to retrieve relevant content. The assistant then augments the context and generates accurate responses based on specialized instructions and the retrieved information. This integrated process is covered in detailed steps below:

1. Environment Setup and API Configuration

Setting up your environment and configuring access to OpenAI's API is the foundational step.

- Create an OpenAI Account.
- Once logged in, navigate to the OpenAI API dashboard. Generate a New Project API Key.
- Depending on your usage and plan, OpenAI may require you to set up billing information. Navigate to the Billing section in the dashboard to add your payment details. Refer to Appendix[7] for cost estimations.

Store your API key securely. A **.env** file is used to securely store environment variables, such as your OpenAI API key.

- Set Up a New Project Folder and Virtual Environment:** First, create a new folder for your project. Ensure that a virtual environment is already set up in this folder, as described in section[4.1.3].

- (b) **Create a .env File:** Inside your new folder, make a file called `.env`. This file will store your OpenAI API key.
- (c) **Add Your API Key:** Open the `.env` file and paste your OpenAI API key in this format:

```
OPENAI_API_KEY=your_openai_api_key_here
```

Be sure to replace with your actual API key.

- (d) **Save the .env File:** After adding your key, save the `.env` file in the same folder where you'll keep your Python files.
- (e) **Install Necessary Python Packages:** To make everything work, you need two tools: `openai` and `python-dotenv`. Open a terminal (or Command Prompt) and run this command to install them:

```
pip install python-dotenv openai
```

If you need specific version of these tools used for the code in GitHub repository, you can install them like this:

```
pip install python-dotenv==1.0.1 openai==1.37.2
```

- (f) **Create the Main Python File:** In the same folder, create a new file called `main.py`. All the code snippets attached in this entire section[4.2.1] should be implemented within this file.

To interact with the OpenAI API and load environment variables, you need to import the necessary libraries. The `dotenv` library will be used to load environment variables from the `.env` file.

Code Example: Import Dependencies

```
import os
import openai
import time
from dotenv import load_dotenv
```

Next, you need to load the environment variables from the `.env` file and set up the OpenAI API **client**. This is important for authenticating your requests to the OpenAI service and setting up the connection to interact with OpenAI's Assistant API.

Code Example: Set OpenAI API Key and LLM

```
# Load environment variables from .env file
load_dotenv()

# Check if OPENAI_API_KEY is set
openai_api_key = os.getenv("OPENAI_API_KEY")
if not openai_api_key:
    raise EnvironmentError("Error: OPENAI_API_KEY is not
                           set in the environment. Please set it in the .env
                           file.")
```



```
# Set OpenAI key and model
openai.api_key = openai_api_key
client = openai.OpenAI(api_key=openai.api_key)
model_name = "gpt-4o" # Any model from GPT series
```

2. Understanding the Problem Domain and Data Requirements

To develop an effective solution for managing and retrieving information, it's important to understand the problem domain and identify the specific data requirements and *not just provide any data*. For a deeper insight into the challenges of handling PDFs, refer to Section[2.3.1]. Given that this paper focuses on working with PDFs, it is important to emphasize the significance of having relevant and clean data within these documents.

Organize the Knowledge Base Files: After selecting the PDF(s) for your external knowledge base, create a folder named **Upload** in the project directory, and place all the selected PDFs inside this folder.

Common Mistakes and Best Practices

Mistake: Irrelevant or inconsistent data Poor-structured data in PDFs downgrades the quality of embeddings generated for Large Language Models (LLMs), hindering them to understand and process the content more accurately.

Best Practice: Ensure data consistency and relevance All PDFs uploaded to the vector store are consistent in format and highly relevant to the problem domain.

Best Practice: Use descriptive file names and metadata Using descriptive file names and adding relevant metadata can help with debugging, maintenance, and retrieval tasks. Files should be named in a way that reflects their content or relevance to the RAG system.

The following Python code defines a function to upload multiple PDF files from a specified directory to OpenAI vector store, which is a common data structure used for storing and querying high-dimensional vectors, often for machine learning and AI applications. It ensures the directory and files are valid before proceeding with the upload and collects and returns the uploaded files' IDs.

NOTE: The function is called to run only when a new vector store is created, meaning it won't upload additional files to an existing vector store. You can modify the logic as needed to suit your requirements.

Also, please be aware that the files are stored on external servers, such as

OpenAI's infrastructure. OpenAI has specific policies regarding data access and usage to protect user privacy and data security. They state that they do not use customer data to train their models unless explicitly permitted by the user. For more details refer:<https://openai.com/policies/privacy-policy/>. Additionally, the data stored can be deleted easily when necessary either via code:<https://platform.openai.com/docs/api-reference/files/delete> or from the user interface by clicking the delete button here: <https://platform.openai.com/storage/files/>.

Code Example: Upload PDF(s) to the OpenAI Vector Store

```
def upload_pdfs_to_vector_store(client, vector_store_id,
                                directory_path):
    try:
        if not os.path.exists(directory_path):
            raise FileNotFoundError(f"Error: Directory '{
                directory_path}' does not exist.")
        if not os.listdir(directory_path):
            raise ValueError(f"Error: Directory '{
                directory_path}' is empty. No files to
                upload.")

        file_ids = {}
        # Get all PDF file paths from the directory
        file_paths = [os.path.join(directory_path, file)
                       for file in os.listdir(directory_path) if file
                       .endswith(".pdf")]

        # Check if there are any PDFs to upload
        if not file_paths:
            raise ValueError(f"Error: No PDF files found
                               in directory '{directory_path}'.")

        # Iterate through each file and upload to vector
        store
        for file_path in file_paths:
            file_name = os.path.basename(file_path)

            # Upload the new file
            with open(file_path, "rb") as file:
                uploaded_file = client.beta.vector_stores.
                    files.upload(vector_store_id=
                        vector_store_id, file=file)
                print(f"Uploaded file: {file_name} with ID
                        : {uploaded_file.id}")
                file_ids[file_name] = uploaded_file.id

        print(f"All files have been successfully uploaded
              to vector store with ID: {vector_store_id}")
```

```

        return file_ids

    except Exception as e:
        print(f"Error uploading files to vector store: {e}")
        return None

```

3. Creating and Managing Vector Stores in OpenAI

OpenAI Vector stores are used to store files for use by the *file search* tool in Assistant API. This step involves initializing a vector store for storing vector embeddings of documents and retrieving them when needed.

Code Example: Initialize a Vector Store for RAG

```

# Get/Create Vector Store
def get_or_create_vector_store(client, vector_store_name):
    if not vector_store_name:
        raise ValueError("Error: 'vector_store_name' is not set. Please provide a valid vector store name.")

    try:
        # List all existing vector stores
        vector_stores = client.beta.vector_stores.list()

        # Check if the vector store with the given name already exists
        for vector_store in vector_stores.data:
            if vector_store.name == vector_store_name:
                print(f"Vector Store '{vector_store_name}' already exists with ID: {vector_store.id}")
                return vector_store

        # Create a new vector store if it doesn't exist
        vector_store = client.beta.vector_stores.create(name=vector_store_name)
        print(f"New vector store '{vector_store_name}' created with ID: {vector_store.id}")

        # Upload PDFs to the newly created vector store (assuming 'Upload' is the directory containing PDFs)
        upload_pdfs_to_vector_store(client, vector_store.id, 'Upload')
        return vector_store

    except Exception as e:

```

```
print(f"Error creating or retrieving vector store:
      {e}")
return None
```

Common Mistakes and Best Practices

Mistake: Ignoring context and query augmentation strategies Relying solely on the vector embeddings without considering query-specific context or augmentation can lead to suboptimal responses.

Best Practice: Augment Queries with Contextual Information Incorporate additional contextual information when forming queries to improve retrieval quality. Using techniques like relevance feedback or pseudo-relevance feedback can help refine search results.^{[[6]][[1]]}

Best Practice: Handle Naming Conflicts Gracefully When creating vector stores, consider adding a timestamp or unique identifier to the vector store name to avoid naming conflicts and make it easier to manage multiple vector stores.

Best Practice: Chunking strategy By default OpenAI uses a *max chunk size tokens of 800* and *chunk overlap tokens of 400* to chunk the file(s) for Vector Stores. Properly sized chunks ensure that each chunk contains a coherent and contextually meaningful piece of information. If chunks are too large, they may contain unrelated content, conversely, if chunks are too small, they may lack sufficient context to be useful. Configure the variables accordingly the PDF(s).

Once the functions to upload PDF file(s) and creating a vector store are defined you can call it to create **Knowledge Base** for your project by providing **vector store name** and store in a *vector store* object as shown below:

Code Example: Creating Vector Store Object

```
vector_store_name = "" # Ensure this is set to a valid
                        name
vector_store = get_or_create_vector_store(client,
                                           vector_store_name)
```

4. Creating Assistant with Specialized Instructions

After setting up the vector store, the next step is to create an AI assistant using the OpenAI API. This assistant will be configured with specialized instructions and tools to perform RAG tasks effectively. Set the **assistant name**, **description** and **instructions** properties accordingly. Refer to the

best practices, if needed you can also play with the **temperature** and **top p** values as per the project needs for random or deterministic responses.

Code Example: Create and Configure Assistant

```
# Get/Create Assistant
def get_or_create_assistant(client, model_name,
                             vector_store_id):

    assistant_name = "" # Ensure this is set to a valid
                        # name
    description = "" # Ensure Purpose of Assistant is set
                    # here
    instructions = "" # Ensure Specialized Instructions
                     # for Assistant and Conversation Structure is set
                     # here)

    try:
        assistants = client.beta.assistants.list()
        for assistant in assistants.data:
            if assistant.name == assistant_name:
                print("AI Assistant already exists with ID
                    : " + assistant.id)
                return assistant

        assistant = client.beta.assistants.create(
            model=model_name,
            name=assistant_name,
            description=description,
            instructions=instructions,
            tools=[{"type": "file_search"}],
            tool_resources={"file_search": {"
                vector_store_ids": [vector_store_id]}},
            temperature=0.7, # Temperature for sampling
            top_p=0.9 # Nucleus sampling parameter
        )
        print("New AI Assistant created with ID:" +
            assistant.id)
        return assistant

    except Exception as e:
        print(f"Error creating or retrieving assistant: {e
            }")
        return None

assistant = get_or_create_assistant(client, model_name,
                                     vector_store.id)
```

Common Mistakes and Best Practices

Best Practice: Ask the Model to adopt a Persona Provide specialized context-Rich *instructions* that guide the assistant on how to handle queries, what tone to use (e.g., formal, friendly), and which domains to prioritize. This ensures the assistant generates more accurate and contextually appropriate responses.

Best Practice: Use inner monologue or Conversation Structure The idea of inner monologue as a part of *instructions* is to instruct the model to put parts of the output into a structured format. This help understand the reasoning process that a model uses to arrive at a final answer.

Best Practice: Fine-Tune Model Parameters Based on Use Case Adjust parameters such as *temperature* (controls randomness) and *top p* (controls diversity) based on the application needs. This can impact the coherence and creativity of the assistant's outputs. For a customer support assistant, a lower temperature may be preferable for consistent responses, while a more creative application might benefit from a higher temperature.

Best Practice: Classifying Queries into Categories For tasks in which lots of independent sets of instructions are needed to handle different cases, it can be beneficial to first classify the type of query and to use that classification to determine which instructions are needed.

5. Creating Conversation Thread

Creating a thread, initializes a context-aware conversation session where the AI assistant can interact with the user, retrieve relevant information from the vector store, and generate responses based on that context. Additionally, *tool resources* can be attached to the Assistant API threads that are made available to the assistant's tools in this thread.

This capability is essentially important when there is a need to use the same AI assistant with different tools for different threads. They can be dynamically managed to suit the requirements for topic-specific threads, reusing the same Assistant across different contexts or overwriting assistant tools for a specific thread.

Code Example: Initialize a thread for conversation

```
# Create thread
thread_conversation = {
    "tool_resources": {
        "file_search": {
            "vector_store_ids": [vector_store.id]
```

```

    }
}

message_thread = client.beta.threads.create(**
    thread_conversation)

```

6. Initiating a Run

A *Run* represents an execution on a thread. This step involves sending user input to the assistant, which then processes it using the associated resources, retrieves information as needed, and returns a response that could include dynamically fetched citations or data from relevant documents.

The following code allows a user to ask questions to an assistant in a loop. It sends the user's question, waits for the assistant to think and respond, and then displays the response word by word. The process repeats until the user types "exit" to quit.

Code Example: Interact with the LLM

```

# Interact with assistant
while True:
    user_input = input("Enter your question (or type 'exit'
        ' to quit): ")

    if user_input.lower() == 'exit':
        print("Exiting the conversation. Goodbye!")
        break

    # Add a message to the thread with the new user input
    message_conversation = {
        "role": "user",
        "content": [
            {
                "type": "text",
                "text": user_input
            }
        ]
    }

    message_response = client.beta.threads.messages.create(
        (thread_id=message_thread.id, **
        message_conversation)

    run = client.beta.threads.runs.create(
        thread_id=message_thread.id,
        assistant_id=assistant.id

```

```

)

response_text = ""
citations = []
processed_message_ids = set()

while True:
    run_status = client.beta.threads.runs.retrieve(run
        .id, thread_id=message_thread.id)
    if run_status.status == 'completed':
        break
    elif run_status.status == 'failed':
        raise Exception(f"Run failed: {run_status.
            error}")
    time.sleep(1)

while True:
    response_messages = client.beta.threads.messages.
        list(thread_id=message_thread.id)

    new_messages = [msg for msg in response_messages.
        data if msg.id not in processed_message_ids]

    for message in new_messages:
        if message.role == "assistant" and message.
            content:
            message_content = message.content[0].text
            annotations = message_content.annotations

            for index, annotation in enumerate(
                annotations):
                message_content.value =
                    message_content.value.replace(
                        annotation.text, f"[{index}]")
                if file_citation := getattr(annotation
                    , "file_citation", None):
                    cited_file = client.files.retrieve
                        (file_citation.file_id)
                    citations.append(f"[{index}] {
                        cited_file.filename}")

            words = message_content.value.split()
            for word in words:
                print(word, end=' ', flush=True)
                time.sleep(0.05)

            processed_message_ids.add(message.id)

    if any(msg.role == "assistant" and msg.content for
        msg in new_messages):

```



```

        break

    time.sleep(1)

    if citations:
        print("\nSources:", ", ".join(citations))
    print("\n")

```

The flexibility of configuring the assistant and thread or assistant-level tools OpenAI's API makes this approach highly versatile for various use cases. By following the steps outlined in this section and adhering to the provided best practices, developers can effectively build a powerful RAG system.

4.2.2 Using Open-Source LLM Model: Llama We will utilize Ollama, an open-source framework that implements the Llama model, to incorporate Llama-based question generation capabilities within our application. By employing Ollama, we can process user input and generate contextually relevant questions directly through the terminal, offering an efficient and scalable solution for natural language processing tasks in a local environment. This integration will enable seamless question generation without relying on external API services, ensuring both privacy and computational efficiency.

1. Install the following Python libraries

```

pip install pymupdf langchain-huggingface faiss-cpu
pip install ollama sentence-transformers sentencepiece
pip install langchain-community

```

2. Converting PDFs to Text Files

Save this script as `pdf_to_text.py`. This script converts PDF files in a given folder into text files. You have to create one folder at the same directory. The folder name should be *Data*. You have to keep your PDF files in this *Data* folder for the further process. Check the GitHub Link of the Code.

Code Example:

```

import os
import fitz # PyMuPDF for reading PDFs

def convert_pdfs_to_text(pdf_folder, text_folder):
    if not os.path.exists(text_folder):
        os.makedirs(text_folder)

    for file_name in os.listdir(pdf_folder):
        if file_name.endswith(".pdf"):
            file_path = os.path.join(pdf_folder, file_name)

```

```

        text_file_name = os.path.splitext(file_name)
            [0] + ".txt"
        text_file_path = os.path.join(text_folder,
            text_file_name)

        with fitz.open(file_path) as doc:
            text = ""
            for page in doc:
                text += page.get_text()

        with open(text_file_path, "w", encoding="utf-8") as text_file:
            text_file.write(text)
        print(f"Converted {file_name} to {
            text_file_name}")

if __name__ == "__main__":
    pdf_folder = "Data"
    text_folder = "DataTxt"
    convert_pdfs_to_text(pdf_folder, text_folder)

```

Functionality: Converts all PDFs in the Data folder to text files and saves them in the DataTxt folder.

3. Creating the FAISS Index

Save this script as `txt_to_index.py`. This script generates a FAISS index from the text files in the DataTxt folder. Check the GitHub link for the Code.

Code Example:

```

import os
from langchain_huggingface import HuggingFaceEmbeddings
from langchain_community.vectorstores import FAISS

def load_text_files(text_folder):
    texts = []
    for file_name in os.listdir(text_folder):
        if file_name.endswith(".txt"):
            file_path = os.path.join(text_folder,
                file_name)
            with open(file_path, "r", encoding="utf-8") as
                file:
                    texts.append(file.read())
    return texts

def create_faiss_index(text_folder, index_path,
    embedding_model='sentence-transformers/all-MiniLM-L6-
v2'):
    texts = load_text_files(text_folder)

```

```

embeddings = HuggingFaceEmbeddings(model_name=
    embedding_model)
vector_store = FAISS.from_texts(texts, embeddings)

vector_store.save_local(index_path)
print(f"FAISS index saved to {index_path}")

if __name__ == "__main__":
    text_folder = "DataTxt"
    index_path = "DataIndex"
    create_faiss_index(text_folder, index_path)

```

Functionality: Creates a FAISS index and saves it in the DataIndex/ folder. Here, `sentence-transformers/all-MiniLM-L6-v2` is a compact, fast transformer model that generates sentence embeddings for tasks like semantic search and text similarity. It's efficient and ideal for quick, accurate text retrieval in applications like FAISS indexing.

4. Setting Up Ollama and Llama 3.1

Before implementing the last script that handles user queries and generates responses using a Large Language Model (LLM), we need to select an open-source LLM. By using Ollama as the model runner, we can easily integrate Llama 3.1 into our system.

(a) Step 1: Download Ollama

To get started with Ollama, follow these steps:

- i. Visit the official Ollama website.
- ii. Choose the version suitable for your operating system (Windows, macOS, or Linux).
- iii. Download and install the appropriate installer from the website.

(b) Step 2: Install Llama 3.1 Using Ollama

After installing Ollama, you can download and install Llama 3.1 by running the following command in your PowerShell or CMD terminal:

```
oLlama pull Llama3.1
```

(c) Step 3: Running Llama 3.1

Once the Llama 3.1 model is installed, you can start using it by running a command similar to this:

```
oLlama run Llama3.1
```

This command runs the Llama 3.1 model, allowing you to ask questions directly in the terminal and interact with the model in real time.

(d) **Step 4: Test OLLama in VS Code**

create a .bat file to avoid typing the full path each time. Open Notepad, add this:

```
@echo off
"C:\path\to\OLLama\oLlama.exe" %*
```

Replace the path with your own. Save the file as oLlama.bat directly in your project folder.

In the VS Code terminal, run the .bat file with the command:

```
.\oLlama.bat run Llama3.1
```

5. **Implementing RAG-Based Question Generation**

Save this script as main.py. This script retrieves relevant documents using FAISS and generates questions based on the retrieved context using OLLama and Llama 3.1. Check the GitHub Code.

Code Example:

```
import os
from langchain_community.vectorstores import FAISS
from langchain_huggingface import HuggingFaceEmbeddings
from langchain.prompts import PromptTemplate
from langchain.chains import RetrievalQA
from langchain_community.llms import OLLama

# Load FAISS index
def load_faiss_index(index_path, embedding_model):
    # Load the FAISS index using the same embedding model
    embeddings = HuggingFaceEmbeddings(model_name=
        embedding_model)
    vector_store = FAISS.load_local(index_path, embeddings
        , allow_dangerous_deserialization=True)
    return vector_store

# Create the RAG system using FAISS and OLLama (Llama 3.1)
def create_rag_system(index_path, embedding_model='
sentence-transformers/all-MiniLM-L6-v2', model_name="
Llama3.1"):
    # Load the FAISS index
    vector_store = load_faiss_index(index_path,
        embedding_model)

    # Initialize the OLLama model (Llama3.1)
    llm = OLLama(model=model_name)

    # Create a more detailed prompt template
    prompt_template = ""
```

You are an expert assistant with access to the following context extracted from documents. Your job is to answer the user's question as accurately as possible, using the context below.

Context:
{context}

Given this information, please provide a comprehensive and relevant answer to the following question:
Question: {question}

If the context does not contain enough information, clearly state that the information is not available in the context provided.

If possible, provide a step-by-step explanation and highlight key details.

"""

Create a template for formatting the input for the model

```
prompt = PromptTemplate(
    input_variables=["context", "question"],
    template=prompt_template
)
```

Create a RetrievalQA chain that combines the vector store with the model

```
qa_chain = RetrievalQA.from_chain_type(
    llm=llm,
    chain_type="stuff",
    retriever=vector_store.as_retriever(),
    chain_type_kwargs={"prompt": prompt}
)
```

```
return qa_chain
```

Function to run the RAG system with a user question

```
def get_answer(question, qa_chain):
    answer = qa_chain.run(question)
    return answer
```

```
if __name__ == "__main__":
```

```
    # Path to the FAISS index directory
    index_path = "DataIndex"
```

```
    # Initialize the RAG system
```

```
    rag_system = create_rag_system(index_path)
```

```
    # Get user input and generate the answer
```

```

while True:
    user_question = input("Ask your question (or type 'exit' to quit): ")
    if user_question.lower() == "exit":
        print("Exiting the RAG system.")
        break
    answer = get_answer(user_question, rag_system)
    print(f"Answer: {answer}")

```

Functionality: The script takes a user query from the terminal. It retrieves relevant documents using FAISS. Then it generates a answer using the retrieved context with Ollama and Llama 3.1.

Common Mistakes and Best Practices

Incompatible embeddings The FAISS index is typically created using a specific embeddings model. If a different embeddings model is used during querying (e.g., a different version of **sentence-transformers**), it may lead to retrieval mismatches or errors. Always ensure that the same model is used both during indexing and querying.

Model version issues Using an incorrect or unsupported model version (e.g., referencing a non-existent version like **Llama 4.5**) can lead to failures during the model loading process. Always verify that the model version you are using is supported and available.

Overly general prompts Prompts that are too broad or generic can result in vague or irrelevant responses from the model. Craft precise and targeted prompts to ensure more accurate and relevant answers.

Ignoring context Language models can generate incorrect or hallucinated responses when the retrieved documents lack the necessary information, leading the model to fill in gaps inaccurately. Always ensure sufficient context is provided in the query.

Memory leaks Extended use of FAISS and Ollama in continuous loops without proper memory management can result in memory leaks, gradually consuming system resources. Monitor memory usage and free up resources after each loop to avoid system slowdowns.

Model re-initialization Reloading or re-initializing models unnecessarily can slow down your system. Reuse initialized models whenever possible to improve system efficiency and reduce overhead.

Ollama (Llama 3.1) is a local language model that runs entirely on the user's machine, ensuring data privacy and faster response times depending on the system's hardware. The accuracy of its outputs depends on the qual-

ity of its training data, and it can be further improved by fine-tuning with domain-specific knowledge. Fine-tuning involves retraining the model with specialized datasets, allowing it to internalize specific organizational knowledge for more precise and relevant responses. This process keeps the model updated and tailored to the user’s needs while maintaining privacy.

5 Preliminary Evaluation of the Guide

5.1 Feedback Process Overview

This experience report underwent an informal evaluation process aimed at gathering feedback for the section: **Using OpenAI’s Assistant API : GPT Series**. [4.2.1] Although the feedback session was not formally structured, it still provided valuable insights that helped validate the ideas presented in this section and refine the guide based on it. The feedback gathered from participants demonstrates that the workshop was successful. A majority of attendees were able to follow the provided guide and successfully implemented their RAG models by the end of the session.

5.2 Participants

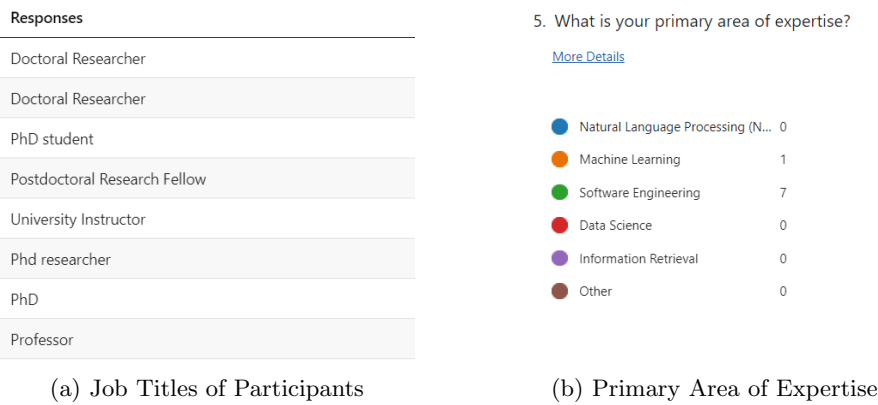


Fig. 3: Demographic Information from Participants

We collected feedback from a small but diverse group of participants during a workshop. A total of 8 individuals completed a demographics form, which provided us with an understanding of the participants’ backgrounds and technical expertise. The group consisted of individuals with varying levels of experience

in machine learning, natural language processing (NLP), and using tools for Retrieval Augmented Generation (RAG). The participants although had familiarity with Python language and OpenAI models.

5.3 Key Feedback Points

During the session, participants shared their thoughts on how much their understanding of RAG systems improve after the workshop, which aspect of the workshop did they find most valuable, challenges they faced and what suggestions or comments they want to provide for future improvement. Below are the key points highlighted by the participants.

8. Before attending the workshop, how familiar were you with Retrieval-Augmented Generation (RAG) systems?

[More Details](#)

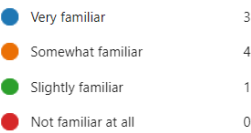


Fig. 4: Participants' Familiarity with RAG Systems.

Prior to attending the workshop, the majority of participants reported a reasonable level of familiarity with RAG systems. This indicated that the audience had a foundational understanding of the concepts presented, allowing for more in depth discussions during the workshop. After the workshop, there was a notable improvement in participants' understanding of RAG systems.

9. How much did your understanding of RAG systems improve after the workshop?

[More Details](#)

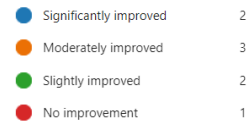


Fig. 5: Participants' Improvement in Understanding RAG Systems.

The majority of participants highlighted the practical coding exercises as the most valuable aspect of the workshop, which helped them better understand the

10. Which aspect of the workshop did you find most valuable?

[More Details](#)

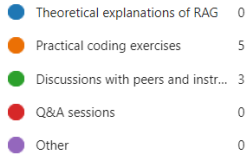


Fig. 6: Most Valuable Aspects of the Workshop.

implementation of RAG systems. Additionally, several participants mentioned the discussions with peers and instructors as a key takeaway.

5.4 Incorporating Feedback to Improve the Guide

The evaluation also revealed opportunities for improvement of guide, particularly in enhancing the clarity of instructions and streamlining the implementation process. The most common issues raised were technical, most of them related to copying from PDF file generating errors as number of lines. In addition to that, we implemented error handling to throw meaningful errors to the user in the code snippets provided to seamlessly run the code.

2	anonymous	Well structured of instructions but copy code should be with terminal command
3	anonymous	Copy-pasting code from the pdf files.
4	anonymous	Some code problems...
5	anonymous	it is very easy to understand
6	anonymous	Havent face any challenges
7	anonymous	Editor and environment issues not relating to the guide.

Fig. 7: Feedback on challenges faced during the implementation of the guide

Several participants also provided suggestions for future improvement. One notable suggestion was to include a warning regarding sensitive data in OpenAI vector store. The detailed comment are shown in Fig[8].

In conclusion, the evaluation process proved valuable in validating the approach outlined in the guide. By testing it in a hands on workshop environment, and in an open discussion sessions of how the developed RAG models improved

12. Any other comments or suggestions for future improvement?

3 Responses

ID ↑	Name	Responses
1	anonymous	My last remark is about the vectoreStore, I just found out it is stored in OpenAI's server, so there should be a warning to not upload any sensitive documents. And also, I want to know , in the guide: where it is stored, and how to delete it.

Fig. 8: Comments and suggestions for improving the guide

trustworthiness in specific scenarios, we were able to collect meaningful feedback and directly address areas of difficulty faced by practitioners. The feedback driven improvements have not only made the guide more user friendly but also demonstrated the importance of continuous iteration based on real world use.

6 Discussion

Practitioners in fields like healthcare, legal analysis, and customer support, often struggle with static models that rely on outdated or limited knowledge. RAG models provide practical solutions with pulling in real time data from provided sources. The ability to explain and trace how RAG models reach their answers also builds trust where accountability and decision making based on real evidence is important.

In this paper, we developed a RAG guide that we tested in a workshop setting, where participants set up and deployed RAG systems following the approaches mentioned. This contribution is practical, as it helps practitioners implement RAG models to address real world challenges with dynamic data and improved accuracy. The guide provides users clear, actionable steps to integrate RAG into their workflows, contributing to the growing toolkit of AI driven solutions.

With that, RAG also opens new research avenues that can shape the future of AI and NLP technologies. As these models and tools improve, there are many potential areas for growth, such as finding better ways to search for information, adapting to new data automatically, and handling more than just text (like images or audio). Recent advancements in tools and technologies have further accelerated the development and deployment of RAG models. As RAG models continue to evolve, several emerging trends are shaping the future of this field.

1. **Haystack:** An open-source framework that integrates dense and sparse retrieval methods with large-scale language models. Haystack supports real-time search applications and can be used to develop RAG models that perform tasks such as document retrieval, question answering, and summarization [4].
2. **Elasticsearch with Vector Search:** Enhanced support for dense vector search capabilities, allowing RAG models to perform more sophisticated retrieval tasks. Elasticsearch's integration with frameworks like Faiss enables

- hybrid retrieval systems that combine the strengths of both dense and sparse search methods, optimizing retrieval speed and accuracy for large datasets[3].
3. **Integration with Knowledge Graphs:** Researchers are exploring ways to integrate RAG models with structured knowledge bases such as knowledge graphs. This integration aims to improve the factual accuracy and reasoning capabilities of the models, making them more reliable for knowledge-intensive tasks[8].
 4. **Adaptive Learning and Continual Fine-Tuning:** There is a growing interest in adaptive learning techniques that allow RAG models to continuously fine-tune themselves based on new data and user feedback. This approach aims to keep models up-to-date and relevant in rapidly changing information environments[7].
 5. **Cross-Lingual and Multimodal Capabilities:** Future RAG models are expected to expand their capabilities across different languages and modalities. Incorporating cross-lingual retrieval and multimodal data processing can make RAG models more versatile and applicable to a wider range of global and multimedia tasks[2].

Future research will likely focus on enhancing their adaptability, cross-lingual capabilities, and integration with diverse data sources to address increasingly complex information needs.

7 Conclusions

The development of Retrieval Augmented Generation (RAG) systems offers a new way to improve large language models by grounding their outputs in real-time, relevant information. This paper covers the main steps for building RAG systems that use PDF documents as the data source. With clear examples and code snippets, it connects theory with practice and highlights challenges like handling complex PDFs and extracting useful text. It also looks at the options available, with examples of using proprietary APIs like OpenAI's GPT and, as an alternative, open-source models like Llama 3.1, helping developers choose the best tools for their needs.

By following the recommendations in this guide, developers can avoid common mistakes and ensure their RAG systems retrieve relevant information and generate accurate, fact-based responses. As technology advances in adaptive learning, multi-modal capabilities, and retrieval methods, RAG systems will play a key role in industries like healthcare, legal research, and technical documentation. This guide offers a solid foundation for optimizing RAG systems and extending the potential of generative AI in practical applications.

References

1. Avi Arampatzis, Georgios Peikos, and Symeon Symeonidis. Pseudo relevance feedback optimization. *Information Retrieval Journal*, 24(4–5):269–297, May 2021.
2. Md Chowdhury, John Smith, Rajesh Kumar, and Sang-Woo Lee. Cross-lingual and multimodal retrieval-augmented generation models. *IEEE Transactions on Multimedia*, 27(2):789–802, 2024.
3. Elasticsearch. Integrating dense vector search in elasticsearch. Elastic Technical Blog, 2023.
4. Haystack. The haystack framework for neural search. Haystack Project Documentation, 2023.
5. Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, and Sebastian Riedel. Retrieval-augmented generation for knowledge-intensive nlp tasks. In *Advances in Neural Information Processing Systems (NeurIPS 2020)*, 2020.
6. Hang Li, Ahmed Mourad, Shengyao Zhuang, Bevan Koopman, and Guido Zuccon. Pseudo relevance feedback with deep language models and dense retrievers: Successes and pitfalls. *ACM Transactions on Information Systems*, 41(3):1–40, April 2023.
7. Percy Liang, Wen-tau Wu, Douwe Kiela, and Sebastian Riedel. Best practices for training large language models: Lessons from the field. *IEEE Transactions on Neural Networks and Learning Systems*, 34(9):2115–2130, 2023.
8. Chenyan Xiong, Zhuyun Dai, Jamie Callan, and Jie Liu. Knowledge-enhanced language models for information retrieval and beyond. *IEEE Transactions on Knowledge and Data Engineering*, 36(5):1234–1247, 2024.

Appendix

1. Tampere University, “Cost Estimation for RAG Application Using GPT-4o”, Zenodo, Sep. 2024. doi: 10.5281/zenodo.13740032.