DocChat: Build a Multi-Agent RAG System

Estimated time needed: 60 minutes

Welcome to DocChat!

Have you ever struggled to extract precise information from long, complex documents? Whether it's a research paper, legal contract, technical report, or environmental study, finding the exact details you need can feel overwhelming.

That's where `DocChat` comes in—a multi-agent RAG tool designed to help you ask questions about your documents and receive fact-checked, hallucination-free answers.

Sure, you could use ChatGPT or DeepSeek to accomplish this task, but when dealing with long documents containing multiple tables, images, and dense text, these models struggle with retrieval and are prone to hallucinations.

They often misinterpret tables, overlook key data hidden in footnotes, or even fabricate citations, as demonstrated below. The problem? These models lack document-aware reasoning and don't verify their responses against structured sources.

That's why DocChat takes a different approach. Instead of relying on a single LLM, it combines multiple AI agents, each with a specific role:

- A Hybrid Retriever that intelligently combines **BM25 keyword search** and **vector embeddings** to retrieve the most relevant passages
- A **Research Agent** that analyzes the retrieved content and generates an initial response
- A **Verification Agent** that cross-checks the response against the original document to detect hallucinations and flag unsupported claims
- A **Self-Correction Mechanism** that re-runs the research step if any contradictions or unsupported statements are found

This multi-step, verification-driven approach ensures that DocChat provides precise, document-grounded answers, even for complex and long-form documents that general-purpose chatbots struggle with. Whether you need to extract specific data points, summarize sections, compare multiple reports, or analyze tables, DocChat is built to help you navigate your documents with confidence.

What you'll learn

In this 60-minute hands-on journey, you'll learn how to:

- Build and deploy a **multi-agent RAG** (retrieval-augmented generation) system that intelligently retrieves, analyzes, and verifies information from complex documents
- Integrate **hybrid retrieval techniques** using BM25 and vector search to improve document understanding and relevance
- Implement a **verification pipeline** that detects hallucinations and ensures factual accuracy in AI-generated responses
- Create an intuitive user interface with **Gradio**, making your document analysis tool accessible to anyone

By the end, you won't just have a working project—you'll have a powerful multi-agent RAG system that outperforms traditional chatbots on long, structured documents.

If you're eager to see how generative AI, multi-agent systems, and RAG pipelines can be combined to tackle real-world information retrieval challenges, this project is for you! Let's explore how it works—and why single-model chatbots fail where DocChat excels.

A quick look at DocChat

What does the DocChat app do?

With DocChat, you can:

- Upload and analyze long documents (PDFs, Word files, text reports) with ease

- Ask questions about the content and get precise, source-backed answers
- Extract specific details from structured documents, even those with tables, figures, and dense text
- Avoid AI hallucinations—every response goes through a verification step to ensure factual correctness
- Receive an alert when your question is irrelevant to the uploaded documents—so you know when the AI cannot confidently answer based on the provided sources
- Retrieve accurate answers even when multiple documents are uploaded—DocChat intelligently finds the right document to reference

In the demo video above, DocChat was tested with two documents. The first is the Google 2024 Environmental Report, a large document spanning 86 pages with numerous images and tables. The second is the DeepSeek-R1 Technical Report, which, while not as extensive as the first, still contains a significant number of diagrams and tables.

As demonstrated, DocChat accurately retrieves relevant answers from both documents. Below, you can see the extracted portions containing the correct information. You can also check these documents out yourselves to verify the information.

> Singapore 2nd Facility PUE
>
> Asia Pacific CFE

Meanwhile, DeepSeek and ChatGPT fail to retrieve the correct information from these documents.

- DeepSeek cannot read entire documents, and therefore cannot retrieve the answer

- Meanwhile, ChatGPT simply hallucinated the numbers.

This demonstrates that DocChat excels at reading uploaded documents and providing accurate answers about their content, outperforming both ChatGPT-4o and DeepSeek R1 in document comprehension.

The next section will walk you through:

- How Hybrid Retrieval ensures that the right information is selected, even across multiple documents
- How the Multi-Agent System via LangGraph prevents hallucinations through fact verification
- How the Self-Correction Mechanism iteratively improves answers when contradictions are found

Let's break down how DocChat works under the hood.

Tips for the best experience - Read-only

Keep the following tips handy and refer to them at any point of confusion throughout the tutorial. Do not worry if they seem irrelevant now. We will go through everything step by step later.

Note: this page is read-only, aiming to teach you how to navigate CloudIDE environment. Do NOT run any code on this page.

- You can choose to follow the `quick version` of building and running the application on the next page if you are in a hurry or you are too excited to test the final result
- Whenever you make changes to a file, be sure to save your work

- At any point throughout the project, if you are lost, click the **Table of Contents** icon on the top-left of the page and navigate to your desired content

- At the end of each section, you will be given the fully updated script for that part. And at the end of the project, you will be prompted to pull the complete codebase of the project as well
- Cloud IDE automatically saves any changes you make to your files immediately. You can also save from the toolbar
- For running the application, always ensure that `app.py` is running in the background before opening the Web Application
- You can run a code block by clicking the `>_` on the bottom-right

```
python app.py
```

- Always ensure that your current directory is `/home/project/docchat`. If you are not in the `docchat` folder, certain code files may fail to run. Use the `cd` command to navigate to the correct location
- Make sure you are accessing the application through port `5000`. Clicking the purple Web Application button will run the app through port 5000 automatically

Web Application

- If you get an error about not being able to access another port (e.g., 8888), just `refresh` the app by clicking the small refresh icon. In case of other errors related to the server, simply refresh the page as well.

- To stop execution of `app.py`, in addition to closing the application tab, press `Ctrl+C` in the terminal
- If you encounter an error running the application or after you enter your desired keyword, try refreshing the app using the button on the top of the application's page. You can try inputting a different query as well
- Typically, using the models provided by Watsonx.ai would require Watsonx credentials, including an API key and a project ID. However, in this lab, these credentials are not needed
- One of the agents uses the Granite 3.1 model, which at the time of writing this lab is unstable and may experience delays in generating responses. If the app takes more than 2-3 minutes to produce an output, press `Ctrl + C` in the terminal to terminate the process and restart the app.

Quick version: Run the final app

In **less than 10 minutes**, you can have a functioning app only by following this page's instructions!

Disclaimer: Ignore this page and skip to the next page for the step-by-step tutorial. Follow this page ONLY if you want to get the final application up and running without getting into the learning material.

First, set up the DocChat application in your development environment. By the end of this step, you'll have the final application running and ready to explore.

Step 1: Clone the project's GitHub repository

To begin, clone the DocChat application repository from GitHub. This repository contains all the source code you'll be working with throughout this project.

Run the following command in the terminal to clone the repository by clicking the run button >_:

```
git clone --no-checkout https://github.com/ibm-developer-skills-network/zzpwx-docchat.git docchat
cd docchat
git checkout 2-final
```

Next, set up a virtual environment to install the dependencies.

Step 2: Set up a Python virtual environment

Initialize a new Python virtual environment to keep required library versions tidy. You can run the snippet directly by clicking the run button >_.

```
python3.11 -m venv venv
source venv/bin/activate
```

Step 3: Install the required libraries

With your virtual environment activated, install all the required libraries via >_. Approximately it will take **3-5 minutes** to install all the required libraries. Feel free to go grab a coffee while you wait!

The following command installs the key libraries that power **DocChat**:

- `docling` – Handles document processing, chunking, and structured data extraction
- `langgraph` – Enables the multi-agent workflow, allowing different AI agents to collaborate efficiently
- `chromadb` – Provides fast, efficient vector search, ensuring accurate document retrieval
- `langchain` **(RAG)** – Facilitates retrieval-augmented generation, integrating retrieval strategies with LLM-based reasoning
- `gradio` - Builds the user-friendly web interface
- `ibm_watsonx_ai` - Interacts with WatsonX models

```
pip install -r requirements.txt
```

Step 4: Run the Gradio app in the terminal

```
python app.py
```

Note: After it runs successfully, you will see a message similar to the following example in the terminal:

Step 5: Congrats! You can now launch the web application

Since the web application is hosted locally on port 5000, click the following button to view the application you've developed.

Web Application

> **Note**: If this "Web Application" button does not work, follow the following picture instructions to launch the application.

Once you launch your application, a window opens, and you should be able to see the application view similar to the following example:

Upload a document and test out the application! After you press the `Submit` button, it takes around 1-2 minutes for the app (our multi-agent system) to analyze the uploaded document and provide output. **If the application fails, refresh it and try again.**

To stop execution of `app.py`, in addition to closing the application tab, press `Ctrl+C` in the terminal.

Step 6: You have completed the project!

Congratulations on successfully building your DocChat application! You now have two options: continue with the rest of this tutorial to dive deeper into the codebase and enhance your understanding, or stop here if you're satisfied with your progress. The choice is yours!

Setting up your development environment

Before you dive into development, set up your project environment in the Cloud IDE. This environment is based on Ubuntu 22.04 and provides all the tools you need to build your AI-driven Flask application.

Note: If you completed the previous Quick version, run the following command in your terminal before proceeding with the rest of the instructions. If not, you can safely disregard this note.

```
cd ..
rm -rf docchat
deactivate
```

Step 1: Create your project directory

1. Open the terminal in Cloud IDE and run the following:

```
git clone --no-checkout https://github.com/ibm-developer-skills-network/zzpwx-docchat.git docchat
cd docchat
git checkout l-start
```

Once you are all set up, select **File Explorer** and then the `docchat` folder. You should have all the files structured as below.

2. Next, set up a virtual environment for the project and install the required libraries.

Note you can run the snippet directly by clicking the run button `>_`. It will take **3-5 minutes** to install all the required libraries. Feel free to go grab a coffee while you wait!

The following command installs the key libraries that power **DocChat**:

- `docling` – Handles document processing, chunking, and structured data extraction
- `langgraph` – Enables the multi-agent workflow, allowing different AI agents to collaborate efficiently
- `chromadb` – Provides fast, efficient vector search, ensuring accurate document retrieval
- `langchain` **(RAG)** – Facilitates retrieval-augmented generation, integrating retrieval strategies with LLM-based reasoning
- `gradio` - Builds the user-friendly web interface
- `ibm_watsonx_ai` - Interacts with WatsonX models

```
python3.11 -m venv venv
source venv/bin/activate # activate venv
pip install -r requirements.txt
```

Now that your environment is set up, you're ready to start building your DocChat!

Understand why multi-agent RAG is used

A Naïve RAG (Retrieval-Augmented Generation) pipeline is often insufficient for handling long, structured documents due to several limitations:

- **Limited query understanding**: Naïve RAG processes queries at a single level, failing to break down complex questions into multiple reasoning steps. This results in shallow or incomplete answers when dealing with multi-faceted queries.

- **No hallucination detection or error handling**: Traditional RAG pipelines lack a verification step. This means that if a response contains hallucinated or incorrect information, there's no mechanism to detect, correct, or refine the output.
- **Inability to handle out-of-scope queries**: Without a proper scope-checking mechanism, Naïve RAG may attempt to generate answers even when no relevant information exists, leading to misleading or fabricated responses.
- **Inefficient multi-document retrieval**: When multiple documents are uploaded, a Naïve RAG system might retrieve irrelevant or suboptimal passages, failing to select the most relevant content dynamically.

To overcome these challenges, DocChat implements a multi-agent RAG research system, which introduces intelligent agents to enhance retrieval, reasoning, and verification.

How multi-agent RAG solves these issues

Scope checking & routing

- A Scope-Checking Agent first determines whether the user's question is relevant to the uploaded documents. If the query is out of scope, DocChat explicitly informs the user instead of generating hallucinated responses.

Dynamic multi-step query processing

- For complex queries, an Agent Workflow ensures that the question is broken into smaller sub-steps, retrieving the necessary information before synthesizing a complete response.
- For example, if a question requires comparing two sections of a document, an agent-based approach recognizes this need, retrieves both parts separately, and constructs a comparative analysis in the final answer.

Hybrid retrieval for multi-document contexts

When multiple documents are uploaded, the Hybrid Retriever (BM25 + Vector Search) ensures that the most relevant document(s) are selected dynamically, improving accuracy over traditional retrieval pipelines.

Fact verification & self-correction

- After an initial response is generated, a Verification Agent cross-checks the output against the retrieved documents.
- If any contradictions or unsupported claims are found, the Self-Correction Mechanism refines the answer before presenting it to the user.

Shared global state for context awareness

- The Agent Workflow maintains a shared state, allowing each step (retrieval, reasoning, verification) to reference previous interactions and refine responses dynamically.
- This enables context-aware follow-up questions, ensuring that users can refine their queries without losing track of previous answers.

Project overview

Below is a breakdown of DocChat's workflow

---

1 - User query processing & relevance analysis

- The system starts when a user submits a question about their uploaded document(s)
- Before retrieving any data, DocChat first analyzes query relevance to determine if the question is within the scope of the uploaded content

2 - Routing & query categorization

- The query is routed through an intelligent agent that decides whether the system can answer it using the document(s):
  - **In scope**: Proceed with document retrieval and response generation.
  - **Not in scope**: Inform the user that the question cannot be answered based on the provided documents, preventing hallucinations.

3 - Multi-agent research & document retrieval

- If the query is relevant, DocChat retrieves relevant document sections from a hybrid search system:
  - Docling converts the document into a structured Markdown format for better chunking
  - LangChain splits the document into logical chunks based on headers and stores them in ChromaDB (a vector store)
  - The retrieval module searches for the most contextually relevant document chunks using BM25 and vector search

4 - Answer generation & verification loop

- Conduct research:
  - The research agent generates an initial answer based on retrieved content
  - A sub-process starts where queries are dynamically generated for more precise retrieval
- Verification process:
  - The verification agent cross-checks the generated response against the retrieved content
  - If the response is fully supported, the system finalizes and returns the answer
  - If verification fails (e.g., hallucinations, unsupported claims), the system re-runs the research step until a verifiable response is found

5 - Response finalization

- After verification is complete, DocChat returns the final response to the user
- The workflow ensures that each answer is sourced directly from the provided document(s), preventing fabrication or unreliable outputs

In the next step, start by building the vector database.

Build vector database

1. Document parsing with Docling

Processing PDFs with **complex structures, tables, and intricate layouts** requires careful selection of a reliable document parsing tool. Many libraries struggle with accuracy when dealing with nested tables, multi-column formats, or scanned PDFs, often resulting in misaligned text, missing data, or broken layouts.

To overcome these challenges, DocChat leverages Docling—an open-source document processing library designed for high-precision parsing and structured data extraction.

Why Docling?

- **Accurate table & layout parsing**: Recognizes complex table structures, reading sequences, and multi-column layouts
- **Multi-format support**: Reads and exports documents in Markdown, JSON, PDF, DOCX, PPTX, XLSX, HTML, AsciiDoc, and images
- **OCR for scanned PDFs**: Extracts text from scanned documents using optical character recognition (OCR)
- **Seamless integration with LangChain**: Enables structured chunking for better retrieval and vector search in ChromaDB

To showcase Docling's superiority over LangChain in handling complex PDFs, we compare both tools on two types of scanned PDF documents: one saved as an image file and another saved as a standard PDF.

Click the button below to inspect the file `test1.py`

Open **test1.py** in IDE

This script compares Docling and LangChain in parsing text from PDF documents. It processes PDFs using both approaches and prints the extracted content for evaluation.

The Docling approach

- Uses `DocumentConverter` to extract structured content
- Converts the PDF into Markdown format
- Splits the extracted content based on headers using `MarkdownHeaderTextSplitter`
- Prints the full extracted sections for review

LangChain approach

- Uses `PyPDFLoader` to load the document
- Extracts raw text from each PDF page
- Prints the entire extracted text

Execute the code below to run the script.

```
python test/test1.py
```

After a few minutes, you should see this output in the terminal:

As demonstrated in the results, Docling successfully extracts text from both scanned file versions, whether saved as an image or as a PDF. In contrast, LangChain's built-in PyPDFLoader fails to extract text from both cases, highlighting a major limitation.

This occurs because PyPDFLoader is designed for digitally generated PDFs that contain embedded text, rather than scanned documents where the text is essentially an image. Since LangChain does not have built-in OCR (Optical Character Recognition) capabilities, it cannot process PDFs that contain scanned images of text, making it ineffective for handling historical documents, academic papers, or other non-digitally generated content.

On the other hand, Docling is equipped to handle both structured and unstructured PDFs, including scanned documents, making it a far more versatile and reliable tool for text extraction in complex scenarios. This demonstrates Docling's advantage in working with real-world PDFs, where many documents are scanned rather than digitally created.

2. Building a vector database with ChromaDB

Once documents have been parsed and structured using **Docling**, the next step is to efficiently store and retrieve relevant document chunks. This is where **ChromaDB** comes into play—a high-performance vector database optimized for fast and accurate similarity search.

What is Chroma DB?

Chroma DB is an open-source vector database optimized for fast and scalable similarity search. It enables efficient storage, retrieval, and ranking of document embeddings, making it a key component of RAG workflows.

Why ChromaDB?

- **Blazing-fast vector search**: Finds the most relevant document chunks in milliseconds
- **Persistent storage**: Keeps embeddings saved for reuse across sessions
- **Seamless LangChain integration**: Works natively with LangChain for retrieval-augmented generation (RAG)
- **Scalable and lightweight**: Handles millions of embeddings efficiently without complex infrastructure

Click the button below to open the `file_handler.py` file, where Docling is used to process uploaded documents.

Open **file_handler.py** in IDE

Then, click the solution below and copy-paste the code into `file_handler.py`

▶ Click for the solution

Overview

The `DocumentProcessor` class is responsible for handling **document parsing, caching, and chunking**. It ensures efficient processing by:

- **Validating file sizes** before processing
- **Using caching** to avoid redundant processing of previously uploaded files
- **Extracting structured content** from documents using **Docling**
- **Splitting text into chunks** using **MarkdownHeaderTextSplitter** for better retrieval in vector databases

---

Function breakdown

`__init__(self)`

- Initializes the **document processor** with:
  - A predefined **header structure** for Markdown-based chunking
  - A **cache directory** for storing processed document chunks
  - Ensures that the **cache directory exists**

`validate_files(self, files: List) -> None`

- **Purpose**: Ensures that the total size of uploaded files does not exceed a predefined limit
- **How it works**:
  - Computes the total size of all uploaded files
  - Compares the total size against `constants.MAX_TOTAL_SIZE`
  - Raises a `ValueError` if the limit is exceeded

`process(self, files: List) -> List`

- **Purpose**: Handles the entire document processing pipeline, including caching and deduplication
- **How it works**:
  - **Validates** the uploaded files
  - **Generates a hash** of each file's content to check if it has been processed before
  - If cached, **loads the data from cache**
  - If not cached, **processes the file** using `_process_file()` and **stores the results in cache**
  - Ensures that **no duplicate chunks** are stored across multiple files

---

`_process_file(self, file) -> List`

- **Purpose**: Converts the document into Markdown and splits it into structured text chunks
- **How it works**:
  - **Skips unsupported file types** (only processes `.pdf`, `.docx`, `.txt`, and `.md`)
  - Uses **Docling's** `DocumentConverter` to convert the file to **Markdown**.
  - Splits the extracted Markdown text using **MarkdownHeaderTextSplitter**.

`_generate_hash(self, content: bytes) -> str`

- **Purpose**: Generates a **unique SHA-256 hash** from document content
- **Use case**: Helps in detecting duplicate files and chunks.

`_save_to_cache(self, chunks: List, cache_path: Path)`

- **Purpose**: Saves the processed document chunks in a **pickle file** for future use
- **How it works**:
  - Stores the `chunks` along with a **timestamp** for expiration checking

---

`_load_from_cache(self, cache_path: Path) -> List`

- **Purpose**: Loads cached document chunks from a previously processed file
- **How it works**:
  - Opens the cached pickle file and **extracts stored document chunks**

`_is_cache_valid(self, cache_path: Path) -> bool`

- **Purpose**: Checks if the cached file is **still valid** (not expired)
- **How it works**:
  - Compares the **modification timestamp** of the cached file against the `CACHE_EXPIRE_DAYS` setting.
  - If the file is **older than the expiration threshold**, it is considered **invalid**.

Quick function recap

Function	Purpose
`__init__()`	Initializes cache directory and header settings.
`validate_files(files: List)`	Ensures that uploaded files do not exceed the size limit.

Function	Purpose
`process(files: List) -> List`	Handles document processing, caching, and deduplication.
`_process_file(file) -> List`	Converts a document into Markdown and splits it into chunks.
`_generate_hash(content: bytes) -> str`	Creates a unique hash of file content.
`_save_to_cache(chunks: List, cache_path: Path)`	Saves processed document chunks to cache.
`_load_from_cache(cache_path: Path) -> List`	Loads cached document chunks if available.
`_is_cache_valid(cache_path: Path) -> bool`	Checks if a cached file is still valid.

Summary

The `DocumentProcessor` class ensures **efficient document parsing and retrieval** by leveraging:

- Docling for structured content extraction
- ChromaDB-compatible chunking for vector search
- A caching system to avoid redundant processing

By using this approach, **DocChat** can efficiently retrieve relevant document chunks, making AI-powered retrieval-augmented generation (RAG) more scalable.

LangGraph multi-agent system structure

Understanding LangGraph: a multi-agent orchestration framework

1. What is agentic AI?

Agentic AI describes a system or program that can independently execute tasks for a user or another system. It autonomously designs workflows, utilizes available tools, makes decisions, takes actions, solves complex problems, and interacts with external environments, extending its capabilities beyond the data used to train its machine learning (ML) models.

2. What is a multi-agent system (MAS)?

A multi-agent system (MAS) is composed of multiple artificial intelligence (AI) agents collaborating to carry out tasks for a user or another system.

3. Key relationship between agentic AI and MAS

Characteristic	Agentic AI	Multi-agent systems (MAS)
Autonomy	Central focus—autonomous task execution	May include agents with varying levels of autonomy
Interaction	Limited to tools, systems, or environments	Key focus—agents interact, communicate, and coordinate
Scope	Individual agent	Multiple agents in a shared system
Dependency	Agentic AI can exist independently	MAS may involve agentic AI but doesn't require it

4. What is LangGraph?

LangGraph is an **open-source Python framework** designed for **multi-agent workflows** in AI applications. It extends **LangChain** by enabling **graph-based state management**, making it easier to coordinate multiple AI agents in structured workflows.

LangGraph is particularly useful in RAG and **multi-step reasoning**, where multiple agents collaborate to refine, verify, and improve responses dynamically.

Key features of LangGraph

- **Graph-based execution**: Workflows are defined as state machines, allowing structured decision-making
- **Multi-agent coordination**: Easily integrates multiple agents, each responsible for a specific task
- **Dynamic state management**: Maintains memory and enables iterative refinement of AI-generated responses
- **Supports loops & conditionals**: Workflows can adapt based on real-time decisions

---

5. How does LangGraph work?

LangGraph operates on the principle of **stateful workflows**, where each step in the process is defined as a **node** in a directed graph. The edges define **transitions** between nodes based on logic.

A LangGraph workflow consists of:

- **Nodes**: They represent individual processing steps (for example, research, verification).
- **Edges**: They define the flow of execution (for example, go to verification after research).
- **State objects**: They store data passed between agents.
- **Conditional transitions** → Allow decision-making between nodes.

---

6. Graph structure for this project

The **AgentWorkflow** class constructs the multi-agent system using **LangGraph's** `StateGraph`, ensuring a structured approach to information retrieval and verification.

Workflow breakdown

1. **Check relevance** – The **RelevanceChecker** determines if the query can be answered based on the retrieved documents.
    - If relevant → Proceed to research
    - If irrelevant → Terminate workflow
2. **Research step** – The **ResearchAgent** generates a draft answer using relevant documents.
3. **Verification step** – The **VerificationAgent** assesses the draft answer for accuracy and relevance.
4. **Decision making** – Based on verification:
    - If the answer lacks support → Re-research and refine
    - If verified → End workflow

LangGraph implementation

Click the button below to open the `workflow.py` file.

Open **workflow.py** in IDE

Then, click the solution below and copy-paste the code into `workflow.py`

▶ Click for the solution

In the code, the `StateGraph` defines the workflow structure:

- **Nodes** represent different stages (`check_relevance`, `research`, `verify`).
- **Edges** define transitions based on conditions.

```
workflow = StateGraph(AgentState)
# Define workflow nodes
workflow.add_node("check_relevance", self._check_relevance_step)
workflow.add_node("research", self._research_step)
workflow.add_node("verify", self._verification_step)
# Set entry point
workflow.set_entry_point("check_relevance")
# Define conditional transitions
workflow.add_conditional_edges(
    "check_relevance",
    self._decide_after_relevance_check,
    {
        "relevant": "research",
        "irrelevant": END
    }
)
workflow.add_edge("research", "verify")
workflow.add_conditional_edges(
    "verify",
    self._decide_next_step,
    {
        "re_research": "research",
        "end": END
    }
)
```

You can check other files in `agents/` folder to see how each agent is implemented.

Click the button below to open `relevance_checker.py`:

Open **relevance_checker.py** in IDE

Then, click the solution below and copy-paste the code into `relevance_checker.py`

▶ Click for the solution

Relevance checker: ensuring query-document alignment

The `RelevanceChecker` is responsible for determining whether retrieved documents contain relevant information to answer a given question. It uses an **ensemble retriever** to fetch document chunks and then leverages IBM watsonX AI for classification. The goal is to categorize relevance into three possible labels:

- **"CAN_ANSWER"** – The documents provide sufficient information for a full answer.
- **"PARTIAL"** – The documents mention the topic but lack complete details.
- **"NO_MATCH"** – The documents do not discuss the question at all.

This classification helps filter out irrelevant queries, ensuring that further processing is only performed on useful data.

How it works

1. Retrieving relevant documents

The `check()` method first retrieves document chunks from an ensemble retriever:

```
top_docs = retriever.invoke(question)
if not top_docs:
    return "NO_MATCH"
```

If no documents are retrieved, it immediately classifies the query as "NO_MATCH".

---

2. Constructing the LLM prompt

If relevant documents are found, the system concatenates the `top-k` document chunks into a single string:

```
document_content = "\n\n".join(doc.page_content for doc in top_docs[:k])
```

Then, it generates a prompt for the watsonX AI model:

```
prompt = f"""
You are an AI relevance checker between a user's question and provided document content.
**Instructions:**
- Classify how well the document content addresses the user's question.
- Respond with only one of the following labels: CAN_ANSWER, PARTIAL, NO_MATCH.
...
**Question:** {question}
**Passages:** {document_content}
...
"""
```

- The prompt clearly instructs the AI model to strictly return one of the three labels.
- The AI considers both explicit answers and partial mentions before making a decision.

3. Running the classification model

The `ModelInference` API is used to query IBM watsonX AI:

```
response = self.model.chat(
    messages=[
        {
            "role": "user",
            "content": prompt
        }
    ]
)
```

- The LLM processes the question and passages to determine relevance.

- If an error occurs (e.g., API failure), the function defaults to "NO_MATCH" to prevent crashes.

4. Extracting and validating the LLM response

Once the AI returns a classification:

```
llm_response = response['choices'][0]['message']['content'].strip().upper()
```

- The function ensures that the response is one of the three valid labels:

```
valid_labels = {"CAN_ANSWER", "PARTIAL", "NO_MATCH"}
if llm_response not in valid_labels:
    classification = "NO_MATCH"
else:
    classification = llm_response
```

- If the model returns unexpected text, the function forces "NO_MATCH" to maintain consistency.

Research agent: generating initial responses using document context

Click the button below to open `research_agent.py`:

Open **research_agent.py** in IDE

Then, click the solution below and copy-paste the code into `research_agent.py`

▶ Click for the solution

The `ResearchAgent` is responsible for **generating an initial draft answer** using retrieved documents. It interacts with **IBM watsonX AI** to synthesize responses based on relevant content. This step is crucial in the RAG pipeline, ensuring that AI-generated answers are grounded in the provided data.

Key functions of the research agent

- **Context-aware answer generation**: Produces fact-based responses using retrieved documents
- **Structured prompting**: Ensures that the AI model adheres to precise instructions for accurate outputs
- **Response sanitization**: Cleans and formats LLM responses for better readability

---

How it works

1. Initializing the model

The `ResearchAgent` is initialized with **IBM watsonX ModelInference**, specifically using the `llama-3-2-90b-vision-instruct` model:

```
self.model = ModelInference(
    model_id="meta-llama/llama-3-2-90b-vision-instruct",
    credentials=credentials,
    project_id="skills-network",
    params={
        "max_tokens": 300,    # Controls response length
```

```
            "temperature": 0.3    # Low randomness for consistent answers
        }
    )
```

- The temperature parameter controls the model's creativity.
- Max tokens define how long the generated response can be.

2. Constructing a prompt for the LLM

The agent generates a structured prompt to ensure fact-based and context-aware responses:

```
def generate_prompt(self, question: str, context: str) -> str:
    prompt = f"""
    You are an AI assistant designed to provide precise and factual answers based on the given context.
    **Instructions:**
    - Answer the following question using only the provided context.
    - Be clear, concise, and factual.
    - Return as much information as you can get from the context.

    **Question:** {question}
    **Context:**
    {context}
    **Provide your answer below:**
    """
    return prompt
```

- The AI must rely solely on the retrieved documents (no hallucinations).
- It is explicitly instructed to return as much information as possible while staying factual.

3. Generating a response

The agent aggregates relevant document content before sending it to the AI:

```
context = "\n\n".join([doc.page_content for doc in documents])
prompt = self.generate_prompt(question, context)
```

- Combines multiple document excerpts into a single context block
- Prepares a query tailored for the WatsonX AI model

The LLM is then queried:

```
response = self.model.chat(
    messages=[
        {
            "role": "user",
            "content": prompt
        }
    ]
)
```

- The AI model processes the question + context to generate an informed response.

4. Extracting and cleaning the LLM response

The raw response from the LLM is extracted:

```
llm_response = response['choices'][0]['message']['content'].strip()
```

- If the response is unexpected or malformed, the function returns a fallback message:

```
if not llm_response:
    draft_answer = "I cannot answer this question based on the provided documents."
```

Finally, the response is sanitized for readability:

```
def sanitize_response(self, response_text: str) -> str:
    return response_text.strip()
```

The final output includes both:

- The generated draft answer
- The context used for generation

```
return {
    "draft_answer": draft_answer,
    "context_used": context
}
```

Verification agent: ensuring answer accuracy and relevance

Click the button below to open `verification_agent.py`:

Open **verification_agent.py** in IDE

Then, click the solution below and copy-paste the code into `verification_agent.py`

▶ Click for the solution

The `VerificationAgent` is responsible for **fact-checking and validating generated answers** using the retrieved documents. This agent ensures that the AI-generated response is:

1. **Supported by factual evidence** from the documents
2. **Free from contradictions** or misinformation
3. **Relevant to the original question**

It interacts with **IBM watsonX AI** to analyze the relationship between the answer and its source documents, producing a structured verification report.

How it works

1. Initializing the verification model

The `VerificationAgent` initializes the **IBM watsonX ModelInference** with a predefined configuration:

```
self.model = ModelInference(
    model_id="ibm/granite-3-8b-instruct",
    credentials=credentials,
    project_id="skills-network",
    params={
        "max_tokens": 200,    # Controls response length
        "temperature": 0.0    # Removes randomness for consistency
    }
)
```

- A low temperature (0.0) ensures that responses remain consistent and deterministic.
- The model is specifically tuned for verification and validation tasks.

2. Constructing a verification prompt

To ensure structured validation, the agent formulates a prompt that asks the model to:

- Confirm factual support for the generated answer
- List unsupported claims or contradictions if found
- Determine whether the answer is relevant to the question

```
def generate_prompt(self, answer: str, context: str) -> str:
    prompt = f"""
    You are an AI assistant designed to verify the accuracy and relevance of answers based on the provided context.
    **Instructions:**
    - Verify the following answer against the provided context.
    - Check for:
      1. Direct/indirect factual support (YES/NO)
      2. Unsupported claims (list any if present)
      3. Contradictions (list any if present)
      4. Relevance to the question (YES/NO)
    - Provide additional details where relevant.
    - Respond in the exact format below.
    **Format:**
    Supported: YES/NO
```

```
    Unsupported Claims: [item1, item2, ...]
    Contradictions: [item1, item2, ...]
    Relevant: YES/NO
    Additional Details: [Any extra information]
    **Answer:** {answer}
    **Context:**
    {context}
    **Respond ONLY with the above format.**
    """
    return prompt
```

- This ensures that the AI model follows a structured output format for easy parsing.
- The AI must strictly adhere to the given labels (YES/NO, lists, or additional explanations).

3. Executing the verification check

Once the prompt is generated, the agent combines retrieved document content and queries the AI model:

```
context = "\n\n".join([doc.page_content for doc in documents])
prompt = self.generate_prompt(answer, context)
```

Then, the LLM is called to analyze the context and verify the answer:

```
response = self.model.chat(
    messages=[
        {
            "role": "user",
            "content": prompt
        }
    ]
)
```

- The AI model checks whether the answer is properly supported by the given documents
- If the API call fails, the system defaults to "NO" for support and relevance to prevent false positives

4. Parsing the AI response

Once the AI generates a verification report, it is parsed into a structured dictionary:

```
def parse_verification_response(self, response_text: str) -> Dict:
    lines = response_text.split('\n')
    verification = {}
    for line in lines:
        if ':' in line:
            key, value = line.split(':', 1)
            key = key.strip().capitalize()
            value = value.strip()
            if key in {"Supported", "Unsupported claims", "Contradictions", "Relevant", "Additional details"}:
                if key in {"Unsupported claims", "Contradictions"}:
                    if value.startswith('[') and value.endswith(']'):
                        items = value[1:-1].split(',')
```

```
                items = [item.strip().strip('"').strip("'") for item in items if item.strip()]
                verification[key] = items
            else:
                verification[key] = []
        elif key == "Additional details":
            verification[key] = value
        else:
            verification[key] = value.upper()
```

- This function extracts structured information from the AI's raw response.
- Lists unsupported claims and contradictions separately for clarity.
- Defaults to "NO" if a field is missing or not formatted correctly.

5. Formatting the verification report

The parsed verification response is then formatted into a readable report:

```
def format_verification_report(self, verification: Dict) -> str:
    supported = verification.get("Supported", "NO")
    unsupported_claims = verification.get("Unsupported Claims", [])
    contradictions = verification.get("Contradictions", [])
    relevant = verification.get("Relevant", "NO")
    additional_details = verification.get("Additional Details", "")
    report = f"**Supported:** {supported}\n"
    if unsupported_claims:
        report += f"**Unsupported Claims:** {', '.join(unsupported_claims)}\n"
    else:
        report += f"**Unsupported Claims:** None\n"
    if contradictions:
        report += f"**Contradictions:** {', '.join(contradictions)}\n"
    else:
        report += f"**Contradictions:** None\n"
    report += f"**Relevant:** {relevant}\n"
    if additional_details:
        report += f"**Additional Details:** {additional_details}\n"
    else:
        report += f"**Additional Details:** None\n"
    return report
```

- The verification results are formatted for easy interpretation.
- If no unsupported claims or contradictions are found, "None" is displayed.

Hybrid retriever: combining BM25 and vector search for optimal document retrieval

The `RetrieverBuilder` class implements a **hybrid retrieval system** by combining:

1. **BM25 (Lexical Search)**: Traditional keyword-based retrieval
2. **Vector Search (Embedding-based)**: Semantic retrieval using embeddings

This combination enhances the accuracy of RAG by leveraging the strengths of both approaches.

Why use a hybrid retriever?

- **Improves recall**: Captures both exact keyword matches and semantically similar content
- **Balances precision & relevance**: BM25 retrieves highly precise keyword matches, while vector retrieval finds related concepts
- **Handles misspellings & variations**: Vector embeddings allow for **fuzzy matching** beyond exact keyword searches
- **Optimized for multi-agent systems**: Ensures robust document retrieval before passing data to AI agents

Why is this is essential for RAG?

- Ensures high-quality document retrieval for multi-agent research workflows
- Improves AI response accuracy by providing both keyword-based and semantic matches
- Enhances retrieval diversity, ensuring that no relevant document is overlooked

With hybrid retrieval, the system achieves a balance between precision and recall, ensuring that AI-generated responses are grounded in the most relevant information.

How the hybrid retriever works

Click the button below to open `builder.py`:

Open **builder.py** in IDE

Then, click the solution below and copy-paste the code into `builder.py`

▶ Click for the solution

1. Initializing watsonX embeddings

The retriever initializes **IBM watsonX embeddings** to enable **semantic vector retrieval**:

```
embed_params = {
    EmbedTextParamsMetaNames.TRUNCATE_INPUT_TOKENS: 3,
    EmbedTextParamsMetaNames.RETURN_OPTIONS: {"input_text": True},
}
watsonx_embedding = WatsonxEmbeddings(
    model_id="ibm/slate-125m-english-rtrvr-v2",
    url="https://us-south.ml.cloud.ibm.com",
    project_id="skills-network",
    params=embed_params
)
self.embeddings = watsonx_embedding
```

- watsonX converts documents into dense vector embeddings.
- These embeddings are later used for semantic similarity search.

2. Building the hybrid retriever

The `build_hybrid_retriever()` method constructs a dual-mode retriever using BM25 and vector-based retrieval.

Step 1: Creating a vector store with ChromaDB

```
vector_store = Chroma.from_documents(
    documents=docs,
    embedding=self.embeddings,
    persist_directory=settings.CHROMA_DB_PATH
)
```

- Stores document embeddings using ChromaDB
- Enables fast vector-based similarity search

Step 2: Initializing BM25 retriever

```
bm25 = BM25Retriever.from_documents(docs)
```

- Uses term frequency-inverse document frequency (TF-IDF) scoring
- Ranks documents based on keyword relevance

Step 3: Creating the vector retriever

```
vector_retriever = vector_store.as_retriever(search_kwargs={"k": settings.VECTOR_SEARCH_K})
```

- Retrieves documents based on vector similarity
- Returns top-k most relevant results

Step 4: Combining both retrievers

```
hybrid_retriever = EnsembleRetriever(
    retrievers=[bm25, vector_retriever],
    weights=settings.HYBRID_RETRIEVER_WEIGHTS
)
```

- Merges BM25 and vector retrieval results into a single ranked list
- Uses HYBRID_RETRIEVER_WEIGHTS to adjust the importance of lexical vs. vector search

Defining the main logic of the application

In this step, we will define the main logic of the application in `app.py`, which integrates all the components of the MAS. To get started, click the button below to open `app.py`.

Open **app.py** in IDE

Then, click the solution below and copy-paste the code into `app.py`

▶ Click for the solution

In this section, we will break down the `app.py` file to understand its components and how it connects the DocChat backend to a user-friendly Gradio interface. This file allows users to interact with the DocChat system by uploading document(s), then having a Q&A session about the document(s).

Why use Gradio?

Gradio is a powerful Python library for building web-based interfaces for machine learning and AI models. While we provide a simple Gradio-based interface to start with, customizing the interface further is beyond the scope of this project. You are welcome to explore more advanced Gradio features to enhance the user experience.

DocChat application: Bringing RAG-based question answering to life

The `app.py` script serves as the **main entry point** for **DocChat**, a **multi-agent RAG (retrieval-augmented generation) system** powered by **Gradio**, **LangGraph**, and **IBM watsonX AI**. It provides a **user-friendly interface** for uploading documents, submitting queries, and retrieving AI-generated answers along with verification reports.

How the application works

The app follows a structured workflow:

1. **Users upload documents** or select a **predefined example**.
2. **The hybrid retriever** extracts relevant document chunks.
3. **The multi-agent system (LangGraph) processes the query**.
4. **The AI generates an answer & a verification report**.
5. **The response is displayed in the Gradio interface**.

Key components in `app.py`

1. Predefined example data

The application provides **example questions and documents** for demonstration:

```
EXAMPLES = {
    "Google 2024 Environmental Report": {
        "question": "Retrieve the data center PUE efficiency values in Singapore...",
        "file_paths": ["examples/google-2024-environmental-report.pdf"]
    },
    "DeepSeek-R1 Technical Report": {
        "question": "Summarize DeepSeek-R1 model's performance...",
        "file_paths": ["examples/DeepSeek Technical Report.pdf"]
    }
}
```

- Users can load these examples instead of manually uploading documents.

2. Initializing core components

The script initializes the three core components responsible for document processing, retrieval, and query handling:

```
processor = DocumentProcessor()
retriever_builder = RetrieverBuilder()
```

```
workflow = AgentWorkflow()
```

- DocumentProcessor: Extracts structured content from uploaded files
- RetrieverBuilder: Constructs a hybrid retrieval system (BM25 + Vector Search)
- AgentWorkflow: Orchestrates the multi-agent processing pipeline using LangGraph

3. Creating the Gradio interface

The app is built using Gradio Blocks, which provides a clean, interactive UI:

```
with gr.Blocks(theme=gr.themes.Citrus(), title="DocChat 🐥", css=css, js=js) as demo:
    gr.Markdown("## DocChat: powered by Docling 🐥 and LangGraph", elem_classes="subtitle")
    gr.Markdown("📥 Upload your document(s), enter your query then press Submit 📨", elem_classes="text")
```

- Uses Markdown descriptions to guide users
- Implements custom CSS & JavaScript for enhanced styling

UI elements

Component	Purpose
`gr.Files()`	Allows users to upload documents
`gr.Textbox()`	Accepts user queries
`gr.Button("Submit 🚀")`	Processes the request and retrieves results
`gr.Dropdown()`	Lets users select predefined example questions
`gr.Button("Load Example 🛠️")`	Loads the selected example into the UI
`gr.Textbox(interactive=False)`	Displays the AI-generated answer and verification report
`gr.Markdown()`	Provides instructions and UI labels

4. Managing document caching & retrieval

To avoid reprocessing unchanged documents, the app maintains a session state:

```
session_state = gr.State({
    "file_hashes": frozenset(),
    "retriever": None
})
```

- When users submit a query, the system checks if the uploaded documents have changed:

```
current_hashes = _get_file_hashes(uploaded_files)
if state["retriever"] is None or current_hashes != state["file_hashes"]:
    logger.info("Processing new/changed documents...")
    chunks = processor.process(uploaded_files)
    retriever = retriever_builder.build_hybrid_retriever(chunks)
    state.update({
        "file_hashes": current_hashes,
        "retriever": retriever
    })
```

- If documents have not changed, the existing retriever will be reused.
- If documents have changed, the system reprocesses them and updates the retriever.

5. Processing queries with the multi-agent workflow

Once the retriever is ready, the LangGraph-based agent system processes the query:

```
result = workflow.full_pipeline(
    question=question_text,
    retriever=state["retriever"]
)
```

- The relevance checker determines if the question can be answered.
- The research agent generates a response using retrieved documents.
- The verification agent validates the response and flags inconsistencies.

Finally, the AI-generated answer and verification report are displayed in the UI.

6. File hashing for efficient processing

To prevent redundant processing, file contents are hashed using SHA-256:

```
def _get_file_hashes(uploaded_files: List) -> frozenset:
    """Generate SHA-256 hashes for uploaded files."""
    hashes = set()
    for file in uploaded_files:
        with open(file.name, "rb") as f:
            hashes.add(hashlib.sha256(f.read()).hexdigest())
    return frozenset(hashes)
```

- This ensures that the same document is not processed multiple times unnecessarily.

7. Launching the application

The Gradio app is deployed locally:

```
demo.launch(server_name="127.0.0.1", server_port=5000, share=True)
```

- Runs on localhost (127.0.0.1) at port 5000.
- The `share=True` flag allows external users to test the application.

Congratulations on completing the implementation of app.py! You've now built a functional interface for the DocChat app, enabling users to chat with uploaded document(s) through a seamless and user-friendly Gradio interface. In the next step, we will launch the application and test it out!

Launching the application

Step 1: Set up the files

Return to the terminal and verify that the virtual environment `venv` label appears at the start of the line. This means that you are in the `venv` environment that you just created. Next, run the following command to pull the complete code of the application.

```
git reset --hard
git checkout 2-final
git pull
```

Step 2: Run the Gradio app in the terminal

```
python app.py
```

Note: After it runs successfully, you will see a message similar to the following example in the terminal:

Step 3: Congrats! You can now launch the web application

Since the web application is hosted locally on port 5000, click the following button to view the application you've developed.

Web Application

Note: If this "Web Application" button does not work, follow the following picture instructions to launch the application.

Once you launch your application. A window opens, and you should be able to see the application view similar to the following example:

Upload an image and test out the application! After you press the `Analyze` button, it takes around 20-30 seconds for the dietary crew (our multi-agent system) to analyze the uploaded image and provide output according to the selected workflow. **If the application fails, refresh it and try again.**

To stop execution of `app.py` in addition to closing the application tab, press `Ctrl+C` in the terminal.

Conclusion and next steps

Conclusion

Congratulations on completing the **DocChat** project!

By building this application, you have successfully explored the power of **multi-agent retrieval-augmented generation (RAG)** using **LangGraph, Docling, watsonX AI, and ChromaDB**. This project integrates multiple components—**document processing, hybrid retrieval, structured AI workflows, and verification agents**—into a **seamless, interactive AI-driven application**.

With the **Gradio-powered UI**, users can intuitively upload documents, ask complex questions, and receive **fact-checked, AI-generated responses** in real time. This marks the culmination of combining **retrieval techniques, structured AI interactions, and LLM-based response generation** into a production-ready system.

What you can do next

Now that you've built the core system, here are some next steps to further enhance your project:

- **Try different embedding models**: Experiment with **OpenAI, Hugging Face, or custom-trained embeddings** to compare retrieval performance
- **Enhance the RAG pipeline**: Improve the **retriever's ranking logic**, adjust **retrieval weights**, or add **post-processing** for better answer formulation
- **Implement Guardrails & AI trust mechanisms**: Use **Llama Guard, AI moderation tools, or manual review processes** to ensure responsible AI usage
- **Optimize the multi-agent workflow**: Tune **verification heuristics, introduce feedback loops, or implement self-improving AI responses**
- **Scale & deploy the app**: Deploy **DocChat** on a **cloud service (e.g., Hugging Face Spaces, AWS, or IBM Cloud)** to make it widely accessible
- **Customize the UI**: Modify the **Gradio interface** to improve user experience, add **chat history tracking**, or enable **document annotations**.

Final thoughts

This project demonstrates how **advanced AI retrieval techniques, multi-agent collaboration, and structured workflows** can be combined to build **real-world, AI-driven applications**. Whether you're enhancing **business intelligence, legal document analysis, or research assistance**, this framework provides **a solid foundation** for creating intelligent document retrieval systems.

Keep experimenting, pushing boundaries, and expanding your AI capabilities—there's so much more to explore!

Author(s)

Hailey Quach

Other Contributor(s)

Ricky Shi
Wojciech "Victor" Fulmyk