Create an AI Agent with RAG in Two Simple Steps

Using the framework Agno, it is simple to augment the agent's capabilities.

. . .

I remember the first time I heard about the word RAG, it sounded like something from another world.

If you also feel that way, you have come to the right place. Stay with me in this tutorial, so we can demystify this concept together.

In this article, we will build an agent that can answer questions about a PDF file by retrieving information from it. Let's dive in.

foto

What is RAG?

Retrieval-Augmented Generation, or just RAG, is nothing more than adding some more information in a *box* that the Large Language Model (LLM) can access and get information from.

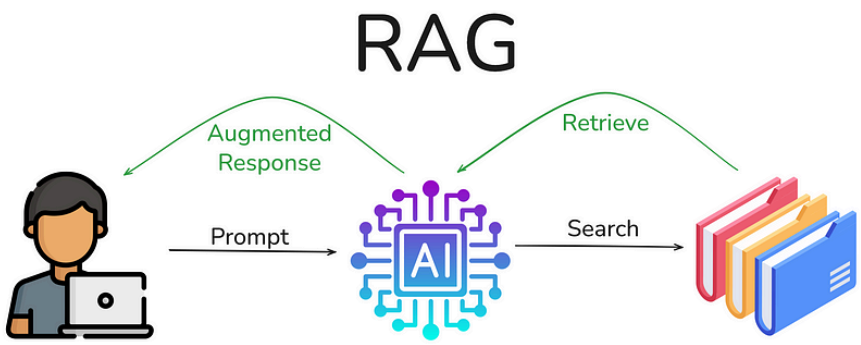
RAG is a store where the LLM can shop for information before returning an answer

Machines can store a lot more information than we do. But they (still) can't store all the information available. Besides being impractical and probably very costly, there is also the problem of information security and new information being generated every single day.

For example, if we ask anything about our daily jobs to an LLM, it will probably say it does not have that information, or it will hallucinate and invent something. That is because the LLM was not trained on proprietary information of your company, so it can't answer questions about something it does not have information about.

Now, what if we could create a folder to serve as the content store to that model and instruct it to access that folder and retrieve information before answering the question?

That is exactly what RAG is about.



RAG workflow. Image by the author.

- The user prompts a question
- The AI goes to the content store
- Searches for the information related to the question
- Retrieves the relevant information
- Enhances the context
- Generates a more accurate response to the user.

Now that we are more familiar with the theory, let's build a RAG.

Code

We will build a simple RAG that lets us *ask questions about a PDF file*.

Our tech stack will be `Agno` for the agent, `Gemini` is the LLM and `Qdrant` is the vector database for the RAG search.

Tech stack for this project. Image by the author.

We are going to do that in two steps, like the title promises. So let's import our modules before we start.

```
# Imports
import os
from agno.agent import Agent
from agno.models.google import Gemini
from agno.knowledge.pdf import PDFKnowledgeBase,
from agno.vectordb.qdrant import Qdrant
from pathlib import Path
```

Step 1: Creating the Content Store

As we previously saw, a RAG needs a content store where it can get information to retrieve and augment the response. This content will be the PDF we will use to ask questions about.

So, first, we create an instance of the **Qdrant** vector database. This type of DB is useful in this case because it stores information by similarity and can return relevant results quickly, integrating very well with LLMs.

We're storing the collection data of the DB in the computer's memory, since it should be small for a RAG for a single PDF file.

```
# Collection name for QDrand
COLLECTION_NAME = "pdf-reader"

# Instance of Qdrant (Vector DB)
vector_db = Qdrant(collection=COLLECTION_NAME, l
```

If you want to learn more about Vector DBs, here's a useful resource.

Vector Databases and Search By Similarity for NLP

Learn about vector databases and how they can help your data science projects.

medium.com

Next, we will use the tool `PDFKnowledgeBase` from Agno's toolkit to create the knowledge base. This tool will:

- Access the PDF file
- Break the text into smaller and relevant chunks
- Store that in the Qdrant instance

```
# Create PDF the knowledge base
pdf_knowledge_base = PDFKnowledgeBase(
    path= Path("path/to/this_article.pdf"),
    vector_db= vector_db,
    reader=PDFReader(chunk=True)
)
```

Nice. I have taken a **PDF print of this article** to serve as an example, so we can ask questions about it to the LLM and assess its performance. It's kind of metalearning.

Now we have our source ready. Let's go to step 2.

Step 2: Creating the Agent

The final step is creating the agent that can access the knowledge base and retrieve the information.

The code is simple. We are using the `Agent` class with `model=Gemini()` and giving instructions to read the document to answer questions.

Then, we point out the knowledge base using the argument `knowledge = pdf_knowledge_base` , and we tell the agent to use it `search_knowledge=True` .

```
# Create agent
agent = Agent(
    model=Gemini(id="gemini-2.0-flash", api_key=
    instructions=[
        "Search your knowledge before answering
        "Only include the content from the agent
        "Only include the output in your respons
    ],
    knowledge=pdf_knowledge_base,
```

```
        search_knowledge=True,
        markdown=True,
```

That's it!

Let us test it.

```
# Test
if __name__ == "__main__":

    # Prompt
    prompt = "What is RAG?"

    # Load the knowledge base, comment out after
    # Set recreate to True to recreate the knowl
    agent.knowledge.load(recreate=False)

    # Run agent
    agent.print_response(
        prompt,
        stream=True,
        show_full_reasoning=True,
        stream_intermediate_steps=True
    )
```

References

Vector Databases and Search By Similarity for NLP

Learn about vector databases and how they can help your data science projects.

medium.com

Qdrant Integration - Agno

Edit description

docs.agno.com

PDF Knowledge Base - Agno

PDFKnowledge is a knowledge base class that allows you to load and query data from PDF files.

docs.agno.com