Explicação do Código

🔧 Importações e Bibliotecas

```
import os
from dotenv import load_dotenv
```

- `os`: biblioteca nativa do Python para interagir com o sistema operacional. Aqui ela serve para pegar variáveis de ambiente, como a chave da API.
- `load_dotenv()`: carrega variáveis guardadas em um arquivo `.env`, evitando expor informações sensíveis no código.

```
from langchain_community.document_loaders import PyMuPDFLoader
```

- `PyMuPDFLoader`: usado para abrir e ler PDFs.
- Cada página do PDF vira um objeto de documento dentro do LangChain, permitindo que o texto seja tratado depois.

```
from langchain_text_splitters import RecursiveCharacterTextSplitter
```

- `RecursiveCharacterTextSplitter`: corta textos grandes em pedaços menores (chunks).
- Isso é importante porque modelos de linguagem têm limite de tokens.
- O "recursivo" significa que ele tenta quebrar respeitando parágrafos ou frases antes de cortar no meio.

```
from langchain_community.vectorstores import FAISS
```

- `FAISS` (Facebook AI Similarity Search): biblioteca para busca vetorial muito rápida.
- Armazena embeddings dos textos e permite encontrar quais trechos são mais próximos de uma pergunta do usuário.

```
from langchain_huggingface import HuggingFaceEmbeddings
```

- `HuggingFaceEmbeddings`: usa modelos do Hugging Face para transformar textos em vetores (embeddings).
- Aqui usamos o `all-MiniLM-L6-v2`, que é leve, rápido e funciona bem para tarefas semânticas.

```
from langchain_openai import ChatOpenAI
```

- `ChatOpenAI`: interface do LangChain compatível com a API da OpenAI.
- O detalhe é que estamos configurando para funcionar via OpenRouter, que fornece acesso a vários modelos diferentes.

```
from langchain.prompts import ChatPromptTemplate
```

- `ChatPromptTemplate`: ajuda a montar prompts de forma organizada, sempre seguindo o mesmo padrão.
- É útil para garantir que a pergunta e o contexto sejam enviados ao modelo de forma consistente.

⚙️ Configurações Iniciais

```
 load_dotenv()
api_key = os.getenv("OPENROUTER_API_KEY")
if not api_key:
    raise ValueError("❌ Variável de ambiente OPENROUTER_API_KEY não encontrada
no .env")
```

- O código carrega o `.env` e verifica se a chave está configurada.
- Caso não esteja, ele já interrompe a execução — isso evita erros silenciosos e garante segurança.

🤖 Instanciando o LLM

```
llm = ChatOpenAI(
    model="mistralai/mistral-7b-instruct:free",
    api_key=api_key,
    base_url="https://openrouter.ai/api/v1",
    temperature=0,
    max_tokens=512,
)
```

- `model`: define qual modelo será usado (aqui o `mistral-7b-instruct` do OpenRouter).
- `temperature=0`: faz com que a resposta seja mais previsível e menos criativa.
- `max_tokens=512`: limite de tamanho da resposta.
- `base_url`: obrigatório para apontar que o modelo vem do OpenRouter.

🧠 Embeddings

```
embeddings = HuggingFaceEmbeddings(
    model_name="sentence-transformers/all-MiniLM-L6-v2"
)
```

- Transforma texto em números (vetores).
- Esses vetores são a "tradução matemática" que permite medir similaridade entre frases.

📄 Carregando o PDF

```
pdf_path = "receitas_bolos.pdf"
if not os.path.exists(pdf_path):
    raise FileNotFoundError(f"❌ Arquivo PDF não encontrado: {pdf_path}")

loader = PyMuPDFLoader(pdf_path)
docs = loader.load()
```

- Garante que o PDF existe.
- Carrega todas as páginas e transforma em documentos de texto manipuláveis.

✂ Divisão em pedaços

```
splitter = RecursiveCharacterTextSplitter(chunk_size=1000, chunk_overlap=200)
splits = splitter.split_documents(docs)
```

- Divide o texto em blocos de até 1000 caracteres.
- Cada bloco tem sobreposição de 200 caracteres com o anterior, para não perder contexto.

📦 Indexação Vetorial

```
vectorstore = FAISS.from_documents(splits, embeddings)
retriever = vectorstore.as_retriever(search_kwargs={"k": 3})
```

- Cria um índice vetorial no FAISS.
- O `retriever` busca os 3 trechos mais parecidos com a pergunta do usuário.

📝 O Prompt

```
prompt = ChatPromptTemplate.from_template(
    "Você é um assistente especializado em receitas de bolos caseiros. ..."
)
```

- Define como o modelo deve se comportar.
- Algumas regras importantes:
  - Sempre responder em português.
  - Só usar informações que estão no PDF.
  - Não inventar nada.
  - Dizer educadamente se a pergunta não tem relação com receitas.

🔄 A Função Principal

```
def responder_pergunta(pergunta: str) -> str:
    ...
```

- Fluxo da função:

1. Recebe a pergunta do usuário.
2. Busca os trechos mais relevantes no índice.
3. Junta esses trechos para formar o contexto.
4. Monta o prompt com a pergunta + contexto.
5. Envia para o LLM via OpenRouter.
6. Retorna a resposta final ao usuário.

🚀 O Fluxo RAG

Esse é o fluxo clássico de RAG (Retrieval-Augmented Generation):

1. O usuário faz uma pergunta.
2. O sistema busca no índice vetorial (FAISS).
3. Constrói o contexto com os trechos recuperados.
4. Envia tudo para o LLM.
5. O modelo responde baseado apenas no documento.
