**Fase 2: Estrutura do Projeto e Código Completo**

Primeiro, organize a sua pasta C:\Users\Usuario\Documents\basf da seguinte forma. Esta estrutura é um padrão de mercado e irá ajudá-lo em projetos futuros.

C:\Users\Usuario\Documents\basf\

├── .venv\ <-- O nosso ambiente virtual isolado

├── assets\ <-- Pasta para ficheiros estáticos como imagens

│ └── Logo\_IAC\_20250519.JPEG

├── .env <-- Ficheiro para as suas chaves de API secretas

├── app.py <-- O código principal da sua aplicação Streamlit

├── manual-de-seguranca-na-aplicacao-de-agroquimicos-2021.pdf

└── requirements.txt <-- Lista de todas as bibliotecas Python necessárias

Agora, vamos criar o conteúdo de cada ficheiro.

**1. Ficheiro requirements.txt**

Este ficheiro lista todas as dependências do projeto.

Plaintext

streamlit

langchain

langchain-openai

langchain\_community

langchain-huggingface

faiss-cpu

pypdf

python-dotenv

**2. Ficheiro .env**

Este ficheiro guardará as suas chaves de API. **Nunca partilhe este ficheiro.**

Fragmento do código

# Substitua pelos seus verdadeiros valores

OPENAI\_API\_KEY="sk-sua-chave-secreta-da-openai"

*Observação: Para este projeto, focaremos no modelo da OpenAI (gpt-4o-mini) pois oferece um excelente equilíbrio entre custo, velocidade e qualidade, sendo ideal para uma aplicação de produção inicial. As outras chaves não são necessárias por agora.*

**3. Ficheiro app.py**

Este é o coração da sua aplicação. É uma versão refinada e otimizada do seu código original, adaptada para execução local e produção.

Python

# -\*- coding: utf-8 -\*-

"""

Agente IA HAMILTON RAMOS - Especialista em Segurança de Agroquímicos

Versão para desenvolvimento local e produção com Streamlit Community Cloud.

"""

# --- 1. Importações de Bibliotecas ---

import streamlit as st

import os

import time

from dotenv import load\_dotenv

# Importações específicas do LangChain

from langchain\_core.messages import AIMessage, HumanMessage

from langchain\_core.prompts import ChatPromptTemplate, MessagesPlaceholder

from langchain\_core.output\_parsers import StrOutputParser

from langchain\_openai import ChatOpenAI, OpenAIEmbeddings

from langchain\_community.document\_loaders import PyPDFLoader

from langchain\_community.vectorstores import FAISS

from langchain\_text\_splitters import RecursiveCharacterTextSplitter

from langchain.chains import create\_history\_aware\_retriever, create\_retrieval\_chain

from langchain.chains.combine\_documents import create\_stuff\_documents\_chain

# --- 2. Configurações Iniciais ---

# Carrega as variáveis de ambiente do ficheiro .env

load\_dotenv()

# Definição de constantes para fácil manutenção

PDF\_PATH = "manual-de-seguranca-na-aplicacao-de-agroquimicos-2021.pdf"

LOGO\_PATH = "assets/Logo\_IAC\_20250519.JPEG"

OPENAI\_MODEL\_NAME = "gpt-4o-mini"

EMBEDDING\_MODEL\_NAME = "text-embedding-3-small"

MODEL\_TEMPERATURE = 0.2

# --- 3. Funções Principais do Agente (com Cache) ---

@st.cache\_resource

def get\_llm():

"""

Cria e retorna uma instância do modelo de linguagem (LLM) da OpenAI.

A anotação @st.cache\_resource garante que o modelo seja carregado apenas uma vez.

"""

api\_key = os.getenv("OPENAI\_API\_KEY")

if not api\_key:

st.error("Chave da API da OpenAI não encontrada. Verifique o seu ficheiro .env.")

return None

try:

return ChatOpenAI(

model=OPENAI\_MODEL\_NAME,

temperature=MODEL\_TEMPERATURE,

openai\_api\_key=api\_key

)

except Exception as e:

st.error(f"Erro ao inicializar o modelo da OpenAI: {e}")

return None

@st.cache\_resource

def get\_vector\_store\_from\_pdf(pdf\_path: str):

"""

Processa o ficheiro PDF, cria os embeddings e armazena-os num vector store (FAISS).

Esta função é intensiva e o cache é crucial para o desempenho.

"""

if not os.path.exists(pdf\_path):

st.error(f"Ficheiro PDF não encontrado em: {pdf\_path}")

return None

try:

with st.spinner(f"A processar o documento '{os.path.basename(pdf\_path)}'... Isto pode demorar um pouco na primeira execução."):

loader = PyPDFLoader(pdf\_path)

documents = loader.load()

text\_splitter = RecursiveCharacterTextSplitter(chunk\_size=1000, chunk\_overlap=200)

splits = text\_splitter.split\_documents(documents)

# Usando embeddings da OpenAI, otimizados para os seus modelos de chat

embeddings = OpenAIEmbeddings(model=EMBEDDING\_MODEL\_NAME)

vector\_store = FAISS.from\_documents(splits, embeddings)

return vector\_store

except Exception as e:

st.error(f"Erro ao processar o PDF e criar o vector store: {e}")

return None

def get\_rag\_chain(\_llm, \_retriever):

"""

Cria e retorna a cadeia RAG completa que processa a pergunta e o histórico.

"""

# 1. Cadeia para reformular a pergunta com base no histórico

contextualize\_q\_system\_prompt = (

"Dada a seguinte conversa e uma pergunta de acompanhamento, reformule a pergunta de acompanhamento "

"para ser uma pergunta independente, que possa ser entendida sem o histórico da conversa. "

"NÃO responda à pergunta, apenas reformule-a se necessário; caso contrário, retorne-a como está."

)

contextualize\_q\_prompt = ChatPromptTemplate.from\_messages([

("system", contextualize\_q\_system\_prompt),

MessagesPlaceholder("chat\_history"),

("human", "{input}"),

])

history\_aware\_retriever = create\_history\_aware\_retriever(

llm=\_llm, retriever=\_retriever, prompt=contextualize\_q\_prompt

)

# 2. Cadeia principal para responder à pergunta com base no contexto recuperado

qa\_system\_prompt = """Você é a IA HAMILTON RAMOS, um assistente virtual especialista em segurança na aplicação de agroquímicos, baseado nos manuais e documentos fornecidos. A sua função é ajudar utilizadores a tirar dúvidas sobre as práticas seguras descritas nesses documentos.

Use APENAS os seguintes trechos dos documentos recuperados para responder à pergunta. Seja claro, objetivo e direto ao ponto.

Se a resposta não estiver nos trechos fornecidos, se a pergunta não for relacionada ao conteúdo dos documentos, ou se você não tiver certeza, diga educadamente que a informação específica não foi encontrada nos documentos fornecidos ou que não pode responder com base no contexto disponível. NÃO invente respostas.

Mantenha a resposta concisa e focada na pergunta.

Responda sempre em português do Brasil.

Contexto Recuperado dos Documentos:

{context}"""

qa\_prompt = ChatPromptTemplate.from\_messages([

("system", qa\_system\_prompt),

MessagesPlaceholder("chat\_history"),

("human", "{input}"),

])

question\_answer\_chain = create\_stuff\_documents\_chain(\_llm, qa\_prompt)

# 3. Combina as duas cadeias numa cadeia de recuperação final

return create\_retrieval\_chain(history\_aware\_retriever, question\_answer\_chain)

# --- 4. Interface Gráfica com Streamlit ---

# Configuração da página

st.set\_page\_config(page\_title="Agente IA HAMILTON RAMOS", page\_icon=LOGO\_PATH)

# Cabeçalho da aplicação

st.image(LOGO\_PATH, width=100)

st.title("Agente IA HAMILTON RAMOS")

st.caption("Especialista em segurança na aplicação de agroquímicos.")

# Inicialização do estado da sessão

if "chat\_history" not in st.session\_state:

st.session\_state.chat\_history = [

AIMessage(content="Olá! Sou a IA HAMILTON RAMOS. Como posso ajudar com as suas dúvidas sobre o manual de segurança?"),

]

# Carregamento do LLM e do Vector Store

llm = get\_llm()

vector\_store = get\_vector\_store\_from\_pdf(PDF\_PATH)

if llm and vector\_store:

# Cria o retriever a partir do vector store

retriever = vector\_store.as\_retriever(search\_type='mmr', search\_kwargs={'k': 5, 'fetch\_k': 8})

# Cria a cadeia RAG

rag\_chain = get\_rag\_chain(llm, retriever)

# Exibição do histórico de chat

for message in st.session\_state.chat\_history:

if isinstance(message, AIMessage):

with st.chat\_message("ai"):

st.markdown(message.content)

elif isinstance(message, HumanMessage):

with st.chat\_message("human"):

st.markdown(message.content)

# Campo de entrada do utilizador

user\_query = st.chat\_input("Digite a sua pergunta aqui...")

if user\_query:

st.session\_state.chat\_history.append(HumanMessage(content=user\_query))

with st.chat\_message("human"):

st.markdown(user\_query)

with st.chat\_message("ai"):

with st.spinner("A pensar..."):

try:

start\_time = time.time()

response = rag\_chain.invoke({

"input": user\_query,

"chat\_history": st.session\_state.chat\_history

})

end\_time = time.time()

answer = response.get('answer', "Desculpe, não consegui gerar uma resposta.")

st.markdown(answer)

# Exibe as fontes utilizadas para a resposta

sources = response.get('context', [])

if sources:

with st.expander(f"Fontes ({len(sources)} trechos consultados em {end\_time - start\_time:.2f}s)"):

for idx, doc in enumerate(sources):

source\_file = os.path.basename(doc.metadata.get('source', 'Desconhecido'))

page = doc.metadata.get('page', 'N/A')

content\_preview = (doc.page\_content[:200] + '...') if len(doc.page\_content) > 200 else doc.page\_content

st.info(f"Fonte {idx+1}: {source\_file} (pág. {page + 1 if isinstance(page, int) else page})\n\"...{content\_preview}...\"")

st.session\_state.chat\_history.append(AIMessage(content=answer))

except Exception as e:

error\_message = f"Ocorreu um erro ao processar a sua pergunta: {e}"

st.error(error\_message)

st.session\_state.chat\_history.append(AIMessage(content=error\_message))

else:

st.warning("O agente não pôde ser inicializado. Verifique as configurações e os ficheiros necessários.")

**Fase 3: Passo a Passo para Execução e Teste Local**

Siga estes passos na sua máquina local, usando o VSCode e o seu terminal.

1. **Crie a Estrutura de Pastas:** Garanta que a sua pasta basf está organizada como descrito acima.
2. **Abra o Terminal no VSCode:** Vá a Terminal > New Terminal.
3. **Crie e Ative um Ambiente Virtual:** Este é um passo crucial para isolar as dependências do seu projeto.

Bash

# Cria o ambiente virtual na pasta .venv

python -m venv .venv

# Ativa o ambiente (Windows)

.\.venv\Scripts\activate

# Ativa o ambiente (macOS/Linux)

# source .venv/bin/activate

Você saberá que funcionou quando vir (.venv) no início da linha do seu terminal.

1. **Instale as Dependências:**

Bash

pip install -r requirements.txt

1. **Preencha as Chaves de API:** Abra o ficheiro .env e insira a sua chave da API da OpenAI.
2. **Execute a Aplicação:**

Bash

streamlit run app.py

O Streamlit irá iniciar um servidor local e abrirá uma aba no seu navegador. A primeira execução pode demorar um pouco mais, pois estará a processar o PDF. As execuções seguintes serão quase instantâneas.

Agora pode interagir com o seu agente localmente e testar tudo!

**Fase 4: Colocando em Produção com Streamlit Community Cloud**

Para partilhar o link com o seu cliente, vamos implementar a aplicação na nuvem.

**Pré-requisitos:**

* Uma conta no [GitHub](https://github.com/).
* Uma conta no [Streamlit Community Cloud](https://share.streamlit.io/) (pode fazer login com a sua conta GitHub).

**Passos:**

1. **Crie um Ficheiro .gitignore:** Na sua pasta basf, crie um ficheiro chamado .gitignore. Isto dirá ao Git para ignorar ficheiros e pastas que não devem ir para o repositório público, como as suas chaves de API.
2. # Ignorar o ambiente virtual
3. .venv/
4. # Ignorar o ficheiro de variáveis de ambiente
5. .env
6. # Ignorar ficheiros de cache do Python
7. \_\_pycache\_\_/
8. \*.pyc
9. **Crie um Repositório no GitHub:**
   * Vá ao GitHub e crie um novo repositório (ex: agente-basf).
   * Siga as instruções para "push an existing repository from the command line".
10. **Envie o seu Código para o GitHub:** No terminal do VSCode:

Bash

git init

git add .

git commit -m "Versão inicial do Agente Hamilton Ramos"

git branch -M main

git remote add origin https://github.com/seu-usuario/agente-basf.git

git push -u origin main

1. **Faça o Deploy no Streamlit Community Cloud:**
   * Vá a [share.streamlit.io](https://share.streamlit.io/) e clique em "New app".
   * Selecione o repositório que acabou de criar.
   * O Streamlit irá detetar o ficheiro app.py automaticamente.
   * **MUITO IMPORTANTE:** Vá à secção "Advanced settings..." e adicione as suas chaves de API como "Secrets".
     + **Secret:** OPENAI\_API\_KEY
     + **Value:** sk-sua-chave-secreta-da-openai
   * Clique em "Deploy!".

O Streamlit irá instalar as dependências do requirements.txt e iniciar a sua aplicação. No final, terá um link público (ex: https://seu-nome-agente-basf.streamlit.app) que pode partilhar com o seu cliente para testes.

**Fase 5: Disponibilizando o Agente no WhatsApp**

Integrar com o WhatsApp é um passo mais avançado que exige uma arquitetura diferente da do Streamlit. Precisamos de um serviço que sirva como ponte entre o WhatsApp e o nosso código Python. A solução mais comum e robusta para isso é a **API do WhatsApp Business com a Twilio**.

**Como Funciona a Arquitetura:**

1. **Utilizador (Agricultor):** Envia uma mensagem no WhatsApp.
2. **Twilio:** Recebe essa mensagem e envia-a para um "webhook" que nós criamos.
3. **Nosso Servidor (Webhook):** É uma aplicação Python (usando um framework como **Flask** ou **FastAPI**) que está sempre online, à espera de receber mensagens da Twilio.
4. **Lógica do Agente:** A nossa aplicação recebe a mensagem, processa-a usando a mesma cadeia RAG que construímos, obtém a resposta da IA.
5. **Resposta à Twilio:** A nossa aplicação envia a resposta de volta para a Twilio.
6. **Twilio para Utilizador:** A Twilio entrega a resposta ao utilizador no WhatsApp.

**Código de Exemplo para um Webhook com Flask e Twilio:**

Para isto, precisaria de adicionar flask e twilio ao seu requirements.txt.

Python

# Ficheiro: whatsapp\_agent.py

# (Este seria um ficheiro separado, não o app.py do Streamlit)

import os

from flask import Flask, request

from twilio.twiml.messaging\_response import MessagingResponse

from dotenv import load\_dotenv

# --- Reutilização da Lógica do Agente ---

# (Aqui importaria as funções do seu app.py, ou melhor, refatoraria

# a lógica do agente para um módulo separado para ser usada por ambos)

# Supondo que a lógica do agente esteja disponível:

# from agent\_logic import get\_agent\_response

load\_dotenv()

app = Flask(\_\_name\_\_)

# Simulação da função de resposta do agente para este exemplo

def get\_agent\_response(query: str, user\_id: str) -> str:

# AQUI, você chamaria a sua cadeia RAG.

# Você também precisaria de gerir o histórico de conversas por utilizador (user\_id).

# Por exemplo, usando um dicionário ou uma base de dados simples.

print(f"Recebida query '{query}' do utilizador {user\_id}")

# response = rag\_chain.invoke({"input": query, "chat\_history": ...})

# return response['answer']

return f"Resposta simulada para a pergunta: '{query}'"

@app.route("/webhook", methods=['POST'])

def webhook():

"""Endpoint que recebe as mensagens do WhatsApp via Twilio."""

# Extrai a mensagem recebida e o número do remetente

incoming\_msg = request.values.get('Body', '').strip()

sender\_id = request.values.get('From', '')

print(f"Mensagem de {sender\_id}: {incoming\_msg}")

# Cria uma resposta TwiML

response = MessagingResponse()

# Obtém a resposta do nosso agente de IA

if incoming\_msg:

agent\_answer = get\_agent\_response(incoming\_msg, sender\_id)

response.message(agent\_answer)

else:

response.message("Não recebi uma mensagem válida. Tente novamente.")

return str(response)

if \_\_name\_\_ == "\_\_main\_\_":

# Para produção, você usaria um servidor WSGI como Gunicorn ou Waitress

app.run(port=5000, debug=True)

**Passos para Implementar com WhatsApp/Twilio:**

1. **Conta Twilio:** Crie uma conta na Twilio e configure o "Twilio Sandbox for WhatsApp".
2. **Desenvolvimento Local com Ngrok:** Para que a Twilio consiga comunicar com a sua máquina local durante o desenvolvimento, você usaria o ngrok para expor a sua aplicação Flask (que corre na porta 5000).

Bash

ngrok http 5000

Isto dar-lhe-á um URL público (ex: https://abcd-1234.ngrok.io).

1. **Configurar Webhook na Twilio:** No painel da Twilio, configure o webhook do Sandbox para apontar para o seu URL do ngrok, seguido de /webhook (ex: https://abcd-1234.ngrok.io/webhook).
2. **Teste:** Envie uma mensagem do seu WhatsApp para o número do Sandbox da Twilio. A mensagem deverá chegar à sua aplicação Flask, e a resposta deverá voltar para o seu WhatsApp.
3. **Produção:** Para uma solução permanente, esta aplicação Flask precisaria de ser hospedada num serviço de nuvem que suporte aplicações Python, como **Render**, **Heroku** ou **AWS Elastic Beanstalk**. A aplicação Streamlit e a aplicação do WhatsApp são independentes e hospedadas em locais diferentes.

A integração com o WhatsApp é um projeto por si só. A minha recomendação é focar primeiro em ter a aplicação Streamlit 100% funcional e validada pelo cliente. Depois, podemos avançar para a integração com o WhatsApp como a "Fase 2" do projeto.

Espero que este guia detalhado lhe dê a clareza e a confiança para levar o seu projeto ao próximo nível. Fez um ótimo trabalho inicial, e com esta estrutura, terá uma solução profissional e escalável.

Estou à sua disposição para esclarecer qualquer dúvida ou para o ajudar nos próximos passos.

Peço desculpas onde eu disse que o projeto está a ser desenvolvido em: C:\Users\Usuario\Documents\basf na verdade é em: C:\Users\Usuario\Documents\basf\gemini

A

Assistente de programação

Gem personalizado

Mostrar pensamento

Sem problema algum! Agradeço imenso o esclarecimento. É um detalhe importante para garantir que todas as instruções e a nossa compreensão do projeto estejam perfeitamente alinhadas.

A boa notícia é que esta mudança é muito simples de incorporar. A lógica, o código e a estrutura interna que definimos permanecem exatamente os mesmos. A única alteração é o "diretório raiz" do seu projeto.

Vamos ajustar as referências para refletir a nova localização: C:\Users\Usuario\Documents\basf\gemini.

**Impacto da Alteração**

* **Código (app.py, etc.):** **Nenhuma alteração necessária.** Como usamos caminhos relativos (ex: assets/Logo... e manual-de-seguranca...), o código continuará a funcionar perfeitamente, desde que seja executado de dentro da pasta gemini.
* **Estrutura de Pastas:** A estrutura que definimos anteriormente será criada *dentro* da pasta gemini.
* **Comandos no Terminal:** Todos os comandos que lhe passei (criar ambiente virtual, instalar dependências, executar a aplicação) devem ser executados com o terminal aberto na pasta gemini.

**Estrutura de Pastas Atualizada**

A sua organização de pastas ficará assim:

C:\Users\Usuario\Documents\basf\

└── gemini\ <-- Este é agora o seu diretório de projeto

├── .venv\

├── assets\

│ └── Logo\_IAC\_20250519.JPEG

├── .env

├── .gitignore

├── app.py

├── manual-de-seguranca-na-aplicacao-de-agroquimicos-2021.pdf

└── requirements.txt

**Passos de Execução (Atualizados)**

Sempre que for trabalhar no projeto, o seu primeiro passo no terminal será navegar para o diretório correto.

1. **Abra o Terminal no VSCode.**
2. **Navegue para a Pasta do Projeto:**

Bash

cd C:\Users\Usuario\Documents\basf\gemini

*Dica: Pode simplesmente abrir a pasta gemini no VSCode e depois abrir um novo terminal. Ele já começará no diretório certo.*

1. **Ative o Ambiente Virtual (se não estiver ativo):**

Bash

.\.venv\Scripts\activate

1. **Execute a Aplicação:**

Bash

streamlit run app.py

Todas as outras instruções, incluindo a criação do repositório no GitHub e a implementação no Streamlit Community Cloud, permanecem idênticas. Apenas lembre-se de que a raiz do seu repositório Git será a pasta gemini.

**Envie o seu Código para o GitHub:** No terminal do VSCode:

git init

git add .

git commit -m "Versão inicial do Agente Hamilton Ramos"

git branch -M main

git remote add origin https://github.com/Icolonia77/agente-basf.git

git push -u origin main

**Opção A: Renomear para main (Recomendado)**

Isso alinha seu projeto com os padrões atuais do GitHub.

Bash

# 1. Renomeia o branch 'master' para 'main'

git branch -M main

# 2. Envia o novo branch 'main' para o GitHub

git push -u origin main

**Opção B: Manter master**

Isso também funciona perfeitamente, mas seu branch principal no GitHub se chamará master.

Bash

# Envia o branch 'master' existente para o GitHub

git push -u origin master