Below is a detailed problem statement, solution, and project idea that leverages generative AI with LangChain, FastAPI, Azure, and other free tools. The project incorporates key LangChain concepts (RAG, chains, vector stores, tools, tool calling) without relying on agents, and it's designed to be cost-free using open-source tools and free tiers. Let's dive in!

Problem Statement

Problem: Individuals and small teams often struggle to efficiently search, summarize, and extract actionable insights from large collections of personal or public documents (e.g., PDFs, web articles, or notes). Existing tools either lack customization, fail to provide context-aware answers, or require expensive subscriptions for advanced features like semantic search or question answering.

Specific Need: Build a web-based application that allows users to:

1. Upload their own documents (e.g., PDFs, text files) or scrape web content.
2. Ask natural language questions about the content.
3. Get concise summaries or detailed answers with citations.
4. Perform real-time calculations or fetch additional web context when needed—all without manual effort.

Example Use Case: A student uploads lecture notes and asks, "What are the key points about photosynthesis?" or "Calculate the energy output based on this formula from the notes."

Project Idea: "DocuQuery" - A Free, AI-Powered Document Q&A and Insight Tool

Project Name: DocuQuery

Tagline: "Ask, Summarize, Compute—Unlock Insights from Your Documents for Free"

Overview: DocuQuery is a web app that uses generative AI to process user-uploaded documents or web-scraped content. It provides a Q&A interface, summarization, and tool-assisted computations,

all powered by LangChain, FastAPI, and Azure’s free tier for hosting. The app avoids paid APIs by using open-source models and local computation.

Solution Design

Here’s how we’ll solve the problem using the specified technologies and LangChain concepts:

- Generative AI:
- Open-source LLM (e.g., LLaMA or Grok via Hugging Face) running locally or via free hosting.
- LangChain: For RAG, chains, vector stores, tools, and output parsing.
- FastAPI: To build a lightweight, async web API for the frontend.
- Azure: Free tier (e.g., Azure App Service or Static Web Apps) for deployment.
- Vector Store: FAISS (free, local vector database) for document embeddings.
- Frontend: Simple HTML/CSS/JavaScript (served via FastAPI or Azure Static Web Apps).
- Free Tools:
- Hugging Face Transformers (for LLM and embeddings).
- Python libraries (e.g., sentence-transformers, PyPDF2).

sentence-transformers

PyPDF2

- Document Upload and Web Scraping: Users upload PDFs/text files or provide URLs to scrape.
- Semantic Search (RAG): Query documents with natural language and get context-aware answers.
- Summarization: Generate concise summaries of documents or sections.
- Tool Calling: Perform calculations or fetch web data based on document content.

- Real-Time Streaming: Display LLM responses as they generate.
- Prompt Templates: Structured prompts for Q&A, summarization, and tool invocation.
- Chains: Sequential and custom chains for processing queries and combining tasks.
- Retrieval-Augmented Generation (RAG): Fetch relevant document chunks for accurate answers.
- Vector Stores: Store document embeddings for fast retrieval.
- Tools and Tool Calling: Custom tools for calculations and web searches.
- Output Parsers: Format LLM responses into clean, actionable outputs.
- Streaming: Real-time response delivery.

Detailed Solution

[User] → [FastAPI Web App]

↘ [Frontend: HTML/JS] ↔ [Backend: FastAPI]

↘ [LangChain Workflow]

- Document Loaders
- Vector Store (FAISS)
- Chains (Sequential, Retrieval)
- Tools (Calculator, Web Search)
- LLM (Local via Hugging Face)

↘ [Azure Free Tier Hosting]

[User] → [FastAPI Web App]

↘ [Frontend: HTML/JS] ↔ [Backend: FastAPI]

↘ [LangChain Workflow]

- Document Loaders
- Vector Store (FAISS)
- Chains (Sequential, Retrieval)

- Tools (Calculator, Web Search)

- LLM (Local via Hugging Face)

↘ [Azure Free Tier Hosting]

- Tools: LangChain PyPDFLoader (for PDFs), WebBaseLoader (for URLs).

PyPDFLoader

WebBaseLoader

- Process:

- User uploads a PDF or enters a URL via the FastAPI endpoint.

- Loader converts content into Document objects with page_content and metadata.

Document

page_content

metadata

- Code Snippet:

```python
from langchain.document_loaders import PyPDFLoader, WebBaseLoader

def load_documents(file_path=None, url=None):
    if file_path:
        loader = PyPDFLoader(file_path)
    elif url:
        loader = WebBaseLoader(url)
    return loader.load()
```

- Tools: sentence-transformers (free embeddings), FAISS (local vector store).

sentence-transformers

- Process:
- Split documents into chunks using RecursiveCharacterTextSplitter.

RecursiveCharacterTextSplitter

- Generate embeddings with HuggingFaceEmbeddings.

HuggingFaceEmbeddings

- Store in FAISS for fast similarity search.
- Code Snippet:

  ```python
  from langchain.text_splitter import RecursiveCharacterTextSplitter
  from langchain.vectorstores import FAISS
  from langchain.embeddings import HuggingFaceEmbeddings

docs = load_documents(file_path="notes.pdf")
splitter = RecursiveCharacterTextSplitter(chunk_size=500, chunk_overlap=50)
chunks = splitter.split_documents(docs)
embeddings = HuggingFaceEmbeddings(model_name="sentence-transformers/all-MiniLM-L6-v2")
vector_store = FAISS.from_documents(chunks, embeddings)
```

- Chains Used:
- SequentialChain: Combines retrieval, Q&A, and summarization.
- RetrievalChain: Fetches relevant chunks from the vector store.
- Process:
- User query → Retrieve top-k chunks → Pass to LLM with a prompt template.

- Code Snippet:

  ```python

  from langchain.prompts import PromptTemplate

  from langchain.chains import LLMChain, SequentialChain

  from langchain.llms import HuggingFacePipeline

llm = HuggingFacePipeline.from_model_id(model_id="meta-llama/Llama-2-7b-hf", task="text-generation")

qa_prompt = PromptTemplate(

  input_variables=["context", "question"],

  template="Given this context: {context}\nAnswer: {question}"

)

qa_chain = LLMChain(llm=llm, prompt=qa_prompt)

```

- Tools:
- Calculator: A Python function to evaluate math expressions.
- Web Search: A mock search using free web scraping (e.g., WebBaseLoader).

WebBaseLoader

- Process:
- LLM decides when to call tools based on the query (e.g., “Calculate 2 + 2” or “What’s the latest on X?”).
- Code Snippet:

  ```python

  from langchain.tools import Tool

def calculate(expression):
  ```

```
    return str(eval(expression))

calc_tool = Tool(name="Calculator", func=calculate, description="Evaluates math expressions")
tools = [calc_tool]

# Manual tool calling logic
if "calculate" in query.lower():
    result = calc_tool.run(query.split("calculate")[1].strip())
```

- Streaming: Use LangChain's streaming support with HuggingFacePipeline.

HuggingFacePipeline

- Output Parser: Parse LLM output into structured format (e.g., JSON).
- Code Snippet:

  ```python
  from langchain.output_parsers import StructuredOutputParser, ResponseSchema

schemas = [ResponseSchema(name="answer", description="The final answer")]
parser = StructuredOutputParser.from_response_schemas(schemas)
```

- Endpoints:
- /upload: Upload document and build vector store.

/upload

- /query: Process user queries with RAG, tools, and streaming.

/query

- Code Snippet:

```python
from fastapi import FastAPI, UploadFile, File
from fastapi.responses import StreamingResponse

app = FastAPI()

@app.post("/upload")
async def upload_file(file: UploadFile = File(...)):
    with open("temp.pdf", "wb") as f:
        f.write(file.file.read())
    docs = load_documents(file_path="temp.pdf")
    # Build vector store
    return {"status": "success"}

@app.get("/query")
async def query_docs(query: str):
    retriever = vector_store.as_retriever()
    context = retriever.get_relevant_documents(query)
    response = qa_chain.run(context=context[0].page_content, question=query)
    return StreamingResponse(iter([response]), media_type="text/plain")
```

- Free Tier: Use Azure App Service (free tier: 60 CPU minutes/day) or Static Web Apps for frontend.
- Steps:
- Push code to GitHub.
- Deploy FastAPI app via Azure CLI: az webapp up --sku F1.

```
az webapp up --sku F1
```

Project Execution Plan

- Setup: Install Python, FastAPI, LangChain, Hugging Face Transformers, FAISS (all free).
- Local Development:
- Build document ingestion and vector store.
- Implement chains and tools.
- Test with sample PDFs (e.g., open-access research papers).
- Web App: Create FastAPI endpoints and a simple frontend.
- Deployment: Host on Azure free tier with GitHub integration.

Why It's Free

- LLM: Local Hugging Face model (no API costs).
- Vector Store: FAISS runs locally.
- Hosting: Azure free tier.
- Dependencies: Open-source libraries.

Sample Output

- Input: Upload a biology PDF, ask "What is photosynthesis?"
- Output: "Photosynthesis is the process by which plants convert light into energy. [Source: Page 5]"
- Input: "Calculate energy from E = mc^2 where m=2, c=3"
- Output: "Energy = 18"

This project, DocuQuery, uses all the LangChain concepts you requested explicitly, stays cost-free, and offers a practical, scalable solution. Ready to pitch it as a personal project? Let me know if you

need help refining it further!