5 REAL GENAI PORTFOLIO PROJECTS

Land an AI Job in 2026

DEEPAK JOSE
@DATASCIENCEBRAIN

🚀 5 REAL GenAI Portfolio Projects to Land Your Dream AI Job in 2026 - Complete Implementation Guide

Author	Ⓓ Deepak Jose
URL	https://www.instagram.com/datasciencebrain

✨ WHAT YOU GET:

📁 PROJECT 1: AI Document Intelligence with RAG

Project 1: AI-Powered Document Intelligence System with RAG

Overview

Build a production-ready document Q&A system using RAG (Retrieval Augmented Generation) that allows users to upload PDFs, process them, and ask intelligent questions about the content.

Tech Stack

- **Backend**: FastAPI (Python)
- **Frontend**: React/Next.js
- **Vector DB**: Pinecone/Chroma
- **LLM**: OpenAI GPT-4 or Anthropic Claude
- **Embedding Model**: OpenAI text-embedding-3-small
- **File Processing**: PyPDF2, LangChain

Architecture Flow

```
User Upload → PDF Processing → Chunking → Embeddings → Vector Store →
Query → Retrieval → LLM Response
```

Starter Code

Backend (FastAPI)

```
# main.py
from fastapi import FastAPI, UploadFile, File, HTTPException
from fastapi.middleware.cors import CORSMiddleware
from pydantic import BaseModel
import os
from typing import List
import openai
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain_community.document_loaders import PyPDFLoader
from langchain_openai import OpenAIEmbeddings
from langchain_community.vectorstores import Chroma
from langchain_openai import ChatOpenAI
from langchain.chains import RetrievalQA
import tempfile

app = FastAPI(title="Document Intelligence API")

app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)

# Initialize components
embeddings = OpenAIEmbeddings(model="text-embedding-3-small")
llm = ChatOpenAI(model="gpt-4-turbo-preview", temperature=0)
vectorstore = None

class QueryRequest(BaseModel):
    question: str

class QueryResponse(BaseModel):
    answer: str
    sources: List[str]

@app.post("/upload")
async def upload_document(file: UploadFile = File(...)):
    """Upload and process PDF document"""
    global vectorstore

    if not file.filename.endswith('.pdf'):
        raise HTTPException(400, "Only PDF files are supported")

    try:
        # Save uploaded file temporarily
        with tempfile.NamedTemporaryFile(delete=False, suffix='.pdf') as tmp_fil
e:
            content = await file.read()
            tmp_file.write(content)
            tmp_path = tmp_file.name

        # Load and split document
        loader = PyPDFLoader(tmp_path)
        documents = loader.load()

        text_splitter = RecursiveCharacterTextSplitter(
            chunk_size=1000,
            chunk_overlap=200,
            length_function=len
        )
        chunks = text_splitter.split_documents(documents)

        # Create vector store
        vectorstore = Chroma.from_documents(
            documents=chunks,
            embedding=embeddings,
            persist_directory="./chroma_db"
        )
        # Clean up temp file
        os.unlink(tmp_path)

        return {
            "message": "Document processed successfully",
            "chunks": len(chunks),
            "filename": file.filename
        }

    except Exception as e:
        raise HTTPException(500, f"Error processing document: {str(e)}")

@app.post("/query", response_model=QueryResponse)
async def query_document(request: QueryRequest):
    """Query the uploaded document"""
    global vectorstore

    if vectorstore is None:
        raise HTTPException(400, "No document uploaded yet")

    try:
        # Create QA chain
        qa_chain = RetrievalQA.from_chain_type(
            llm=llm,
            chain_type="stuff",
            retriever=vectorstore.as_retriever(search_kwargs={"k": 3}),
            return_source_documents=True
        )

        # Get answer
        result = qa_chain({"query": request.question})

        sources = [doc.metadata.get('source', 'Unknown') for doc in result['sour
ce_documents']]

        return QueryResponse(
            answer=result['result'],
            sources=list(set(sources))
        )

    except Exception as e:
        raise HTTPException(500, f"Error querying document: {str(e)}")

@app.get("/health")
async def health_check():
    return {"status": "healthy"}

if __name__ == "__main__":
    import uvicorn
    uvicorn.run(app, host="0.0.0.0", port=8000)
```

Frontend (React/Next.js)

```
// app/page.tsx
'use client';

import { useState } from 'react';

export default function Home() {
  const [file, setFile] = useState<File | null>(null);
  const [question, setQuestion] = useState('');
  const [answer, setAnswer] = useState('');
  const [loading, setLoading] = useState(false);
  const [uploaded, setUploaded] = useState(false);

  const handleUpload = async () => {
    if (!file) return;

    setLoading(true);
    const formData = new FormData();
    formData.append('file', file);
    try {
      const response = await fetch('http://localhost:8000/upload', {
        method: 'POST',
        body: formData,
      });

      if (response.ok) {
        setUploaded(true);
        alert('Document uploaded successfully!');
      }
    } catch (error) {
      alert('Error uploading document');
    } finally {
      setLoading(false);
    }
  };

  const handleQuery = async () => {
    if (!question) return;

    setLoading(true);
    try {
      const response = await fetch('http://localhost:8000/query', {
        method: 'POST',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify({ question }),
      });

      const data = await response.json();
      setAnswer(data.answer);
    } catch (error) {
      alert('Error querying document');
    } finally {
      setLoading(false);
    }
  };

  return (
    <div className="min-h-screen bg-gray-50 p-8">
      <div className="max-w-4xl mx-auto">
        <h1 className="text-4xl font-bold mb-8">AI Document Intelligence</h1
>

        {/* Upload Section */}
        <div className="bg-white p-6 rounded-lg shadow mb-6">
          <h2 className="text-2xl font-semibold mb-4">Upload Document</h2
>
          <input
            type="file"
            accept=".pdf"
            onChange={(e) ⇒ setFile(e.target.files?.[0] || null)}
            className="mb-4 block w-full"
          />
          <button
            onClick={handleUpload}
            disabled={!file || loading}
            className="bg-blue-600 text-white px-6 py-2 rounded hover:bg-blue
-700 disabled:bg-gray-400"
          >
            {loading ? 'Uploading...' : 'Upload PDF'}
          </button>
        </div>

        {/* Query Section */}
        {uploaded && (
          <div className="bg-white p-6 rounded-lg shadow">
            <h2 className="text-2xl font-semibold mb-4">Ask Questions</h2>
            <textarea
              value={question}
              onChange={(e) ⇒ setQuestion(e.target.value)}
              placeholder="What would you like to know about this document?"
          className="w-full border rounded p-3 mb-4 h-24"
        />
        <button
          onClick={handleQuery}
          disabled={loading}
          className="bg-green-600 text-white px-6 py-2 rounded hover:bg-g
reen-700 disabled:bg-gray-400"
        >
          {loading ? 'Thinking...' : 'Ask Question'}
        </button>

        {answer && (
          <div className="mt-6 p-4 bg-gray-50 rounded">
            <h3 className="font-semibold mb-2">Answer:</h3>
            <p className="text-gray-700">{answer}</p>
          </div>
        )}
      </div>
    )}
   </div>
  </div>
 );
}
```

Implementation Guide

Step 1: Environment Setup

```
# Create project directory
mkdir rag-document-intelligence
cd rag-document-intelligence

# Backend setup
mkdir backend
cd backend
python -m venv venv
```

```
source venv/bin/activate  # Windows: venv\Scripts\activate
pip install fastapi uvicorn openai langchain langchain-community langchain-o
penai chromadb pypdf2 python-multipart

# Frontend setup
cd ..
npx create-next-app@latest frontend --typescript --tailwind --app
cd frontend
npm install
```

Step 2: Environment Variables

```
# backend/.env
OPENAI_API_KEY=your_openai_api_key_here
```

Step 3: Run the Application

```
# Terminal 1 - Backend
cd backend
uvicorn main:app --reload

# Terminal 2 - Frontend
cd frontend
npm run dev
```

Step 4: Testing

- Upload a PDF document
- Ask questions like "What is the main topic?", "Summarize the key points"

Key Features to Highlight

- Vector similarity search with embeddings
- Chunking strategy for large documents
- Retrieval-augmented generation

- Context-aware responses
- Source attribution

Project 2: Multi-Agent AI Research Assistant with LangGraph

Overview

Build an intelligent research assistant that uses multiple AI agents to search the web, analyze information, and generate comprehensive research reports.

Tech Stack

- **Backend**: FastAPI
- **Agent Framework**: LangGraph
- **LLM**: OpenAI GPT-4
- **Tools**: Tavily Search API, Wikipedia API
- **Frontend**: Streamlit

Architecture Flow

```
User Query → Planner Agent → Search Agent → Analyzer Agent → Writer Age
nt → Final Report
```

Starter Code

Backend (LangGraph Agent System)

```
# agents.py
from typing import TypedDict, Annotated, Sequence
from langchain_core.messages import BaseMessage, HumanMessage, AIMes
sage
from langchain_openai import ChatOpenAI
from langchain_community.tools.tavily_search import TavilySearchResults
from langgraph.graph import StateGraph, END
from langgraph.prebuilt import ToolExecutor
import operator

# Define agent state
class AgentState(TypedDict):
    messages: Annotated[Sequence[BaseMessage], operator.add]
    query: str
    search_results: str
    analysis: str
    final_report: str

# Initialize LLM and tools
llm = ChatOpenAI(model="gpt-4-turbo-preview", temperature=0.7)
search_tool = TavilySearchResults(max_results=5)
tool_executor = ToolExecutor([search_tool])

# Planner Agent
def planner_node(state: AgentState) → AgentState:
    """Plans the research approach"""
    messages = state["messages"]
    query = state["query"]

    system_message = """You are a research planner. Break down the user's q
uery into
    specific search queries that will help gather comprehensive information.
    Return 3-5 specific search queries."""

    response = llm.invoke([
        {"role": "system", "content": system_message},
        {"role": "user", "content": f"Create search plan for: {query}"}
    ])

    return {
        **state,
        "messages": messages + [response],
        "search_queries": response.content
    }

# Search Agent
def search_node(state: AgentState) → AgentState:
    """Executes web searches"""
    query = state["query"]

    # Perform search
    search_results = search_tool.invoke({"query": query})

    # Format results
    formatted_results = "\n\n".join([
        f"Source: {result['url']}\nContent: {result['content']}"
        for result in search_results
    ])

    return {
        **state,
        "search_results": formatted_results
    }

# Analyzer Agent
def analyzer_node(state: AgentState) → AgentState:
    """Analyzes search results"""
    search_results = state["search_results"]
    query = state["query"]

    system_message = """You are a research analyst. Analyze the search result
s and extract:
    1. Key findings
    2. Important statistics or data
    3. Different perspectives
    4. Credibility assessment"""

    response = llm.invoke([
        {"role": "system", "content": system_message},
        {"role": "user", "content": f"Query: {query}\n\nSearch Results:\n{search_r
esults}"}
    ])

    return {
        **state,
        "analysis": response.content
    }

# Writer Agent
def writer_node(state: AgentState) → AgentState:
    """Generates final report"""
    query = state["query"]
    analysis = state["analysis"]

    system_message = """You are a research writer. Create a comprehensive,
well-structured
    research report with:
    1. Executive Summary
    2. Main Findings
    3. Detailed Analysis
    4. Conclusion
    Use markdown formatting."""

    response = llm.invoke([
        {"role": "system", "content": system_message},
        {"role": "user", "content": f"Query: {query}\n\nAnalysis:\n{analysis}"}
    ])

    return {
        **state,
        "final_report": response.content
    }

# Build the graph
```

```
workflow = StateGraph(AgentState)

# Add nodes
workflow.add_node("planner", planner_node)
workflow.add_node("search", search_node)
workflow.add_node("analyzer", analyzer_node)
workflow.add_node("writer", writer_node)

# Add edges
workflow.set_entry_point("planner")
workflow.add_edge("planner", "search")
workflow.add_edge("search", "analyzer")
workflow.add_edge("analyzer", "writer")
workflow.add_edge("writer", END)

# Compile
research_graph = workflow.compile()

# Main research function
async def conduct_research(query: str) → str:
    """Run the multi-agent research workflow"""
    initial_state = {
        "messages": [],
        "query": query,
        "search_results": "",
        "analysis": "",
        "final_report": ""
    }

    result = await research_graph.ainvoke(initial_state)
    return result["final_report"]
```

FastAPI Server

```
# main.py
from fastapi import FastAPI, HTTPException
from fastapi.middleware.cors import CORSMiddleware
from pydantic import BaseModel
from agents import conduct_research

app = FastAPI(title="AI Research Assistant")

app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)

class ResearchRequest(BaseModel):
    query: str

class ResearchResponse(BaseModel):
    report: str

@app.post("/research", response_model=ResearchResponse)
async def create_research(request: ResearchRequest):
    """Generate research report"""
    try:
        report = await conduct_research(request.query)
        return ResearchResponse(report=report)
    except Exception as e:
        raise HTTPException(500, f"Research failed: {str(e)}")

@app.get("/health")
async def health():
    return {"status": "healthy"}
```

Streamlit Frontend

```
# app.py
import streamlit as st
import requests
import time

st.set_page_config(page_title="AI Research Assistant", page_icon="🔍", layou
t="wide")

st.title("🔍 Multi-Agent AI Research Assistant")
st.markdown("Get comprehensive research reports powered by AI agents")

# Sidebar
with st.sidebar:
    st.header("About")
    st.markdown("""
    This assistant uses multiple AI agents:
    - **Planner**: Creates research strategy
    - **Search**: Gathers information
    - **Analyzer**: Processes findings
    - **Writer**: Generates report
    """)

# Main interface
query = st.text_area(
    "What would you like to research?",
    placeholder="E.g., 'Latest developments in quantum computing' or 'Impact
of AI on healthcare'",
    height=100
)

if st.button("Generate Research Report", type="primary"):
    if not query:
        st.warning("Please enter a research query")
    else:
        with st.spinner("AI agents are researching... This may take 1-2 minutes"):
            try:
                response = requests.post(
                    "http://localhost:8000/research",
                    json={"query": query},
                    timeout=180
                )

                if response.status_code == 200:
                    report = response.json()["report"]

                    st.success("Research complete!")
                    st.markdown("---")
                    st.markdown(report)

                    # Download button
                    st.download_button(
                        label="Download Report",
                        data=report,
                        file_name="research_report.md",
                        mime="text/markdown"
                    )
                else:
                    st.error("Failed to generate report")
            except Exception as e:
                st.error(f"Error: {str(e)}")

# Examples
with st.expander("Example Queries"):
    st.markdown("""
    - Latest AI trends in 2024
    - Climate change solutions and innovations
    - Cryptocurrency regulation landscape
    - Gene therapy breakthroughs
    - Future of remote work technology
    """)
```

Implementation Guide

Step 1: Setup

```
mkdir multi-agent-research
cd multi-agent-research

python -m venv venv
source venv/bin/activate
pip install fastapi uvicorn langchain langchain-openai langgraph tavily-python
streamlit
```

Step 2: Environment Variables

```
# .env
OPENAI_API_KEY=your_openai_key
TAVILY_API_KEY=your_tavily_key
```

Step 3: Run

```
# Terminal 1 - Backend
uvicorn main:app --reload

# Terminal 2 - Frontend
streamlit run app.py
```

Key Features

- Multi-agent collaboration
- State management with LangGraph
- Web search integration
- Structured report generation