Python Code

```python
#!/usr/bin/env python
"""
FastAPI Resume Management Server - AGENTIC VERSION
Uses LangChain Agent for autonomous task handling
"""

from fastapi import FastAPI, File, UploadFile, HTTPException
from fastapi.middleware.cors import CORSMiddleware
from fastapi.staticfiles import StaticFiles
from pydantic import BaseModel
from typing import List, Optional
import os
import aiofiles
from datetime import datetime
from dotenv import load_dotenv
import shutil

# Load environment variables
load_dotenv()

if not os.getenv("OPENAI_API_KEY"):
    raise ValueError("OPENAI_API_KEY not found in .env file")

# LangChain imports - using only stable imports
from langchain_openai import ChatOpenAI, OpenAIEmbeddings
from langchain_community.vectorstores import FAISS
from langchain_text_splitters import RecursiveCharacterTextSplitter
from langchain_community.document_loaders import TextLoader, DirectoryLoader, PyPDFLoader
from langchain_core.tools import tool
from langchain_core.messages import HumanMessage, AIMessage, SystemMessage, ToolMessage

# ============================================================================
# CONFIG
# ============================================================================

class Config:
    RESUMES_DIR = "./resumes"
    DB_DIR = "./resume_db"

    @staticmethod
    def setup():
        os.makedirs(Config.RESUMES_DIR, exist_ok=True)
        os.makedirs(Config.DB_DIR, exist_ok=True)

# ============================================================================
# GLOBAL EMBEDDINGS
# ============================================================================

embeddings = OpenAIEmbeddings(model="text-embedding-3-small")

# ============================================================================
# RESUME MANAGEMENT FUNCTIONS
# ============================================================================

def ingest_resumes():
    """Load resumes from ./resumes folder and add to vector database"""
    print("📥 Ingesting resumes...")

    txt_loader = DirectoryLoader(Config.RESUMES_DIR, glob="**/*.txt", loader_cls=TextLoader)
    txt_docs = txt_loader.load()

    pdf_loader = DirectoryLoader(Config.RESUMES_DIR, glob="**/*.pdf", loader_cls=PyPDFLoader)
    pdf_docs = pdf_loader.load()

    all_docs = txt_docs + pdf_docs

    if not all_docs:
        return "No resumes found in ./resumes folder"

    text_splitter = RecursiveCharacterTextSplitter(chunk_size=1000, chunk_overlap=200)
    chunks = text_splitter.split_documents(all_docs)

    db_file_exists = os.path.exists(f"{Config.DB_DIR}/index.faiss")

    if db_file_exists:
        vectorstore = FAISS.load_local(Config.DB_DIR, embeddings, allow_dangerous_deserialization=True)
        vectorstore.add_documents(chunks)
        result = f"Added {len(chunks)} chunks from {len(all_docs)} resumes to existing database"
    else:
        vectorstore = FAISS.from_documents(chunks, embeddings)
        result = f"Created new database with {len(chunks)} chunks from {len(all_docs)} resumes"

    vectorstore.save_local(Config.DB_DIR)
    print(f"✓ {result}")
    return result

def list_resumes():
    """List all resumes stored in vector database"""
    print("📋 Listing resumes...")
    if not os.path.exists(f"{Config.DB_DIR}/index.faiss"):
        return "No database found. Please ingest resumes first."

    vectorstore = FAISS.load_local(Config.DB_DIR, embeddings, allow_dangerous_deserialization=True)
    all_docs = vectorstore.docstore._dict

    sources = set()
    for doc in all_docs.values():
        if hasattr(doc, 'metadata') and 'source' in doc.metadata:
            sources.add(os.path.basename(doc.metadata['source']))

    result = f"Found {len(sources)} resumes in database:\n"
    for i, source in enumerate(sorted(sources), 1):
        result += f"{i}. {source}\n"

    return result

def search_resumes(skills):
    """Search resumes by skills and return best matches"""
    print(f"🔍 Searching for candidates with skills: {skills}")
    if not os.path.exists(f"{Config.DB_DIR}/index.faiss"):
        return "No database found. Please ingest resumes first."

    vectorstore = FAISS.load_local(Config.DB_DIR, embeddings, allow_dangerous_deserialization=True)

    retriever = vectorstore.as_retriever(search_kwargs={"k": 5})
    docs = retriever.invoke(skills)

    context = "\n\n".join([f"Resume {i+1}:\n{doc.page_content}" for i, doc in enumerate(docs)])

    prompt = f"""You are a recruiter assistant. Based on the following resume excerpts, identify and rank the best candidates for the required skills.

Required Skills: {skills}

Resume Excerpts:
{context}

Please provide a summary for the top 3 best matching candidates. For each candidate, include:
- **Candidate Name** (if available, otherwise use "Candidate #")
- **Relevant Skills:** (comma-separated list)
- **Why They Are a Good Fit:** (brief explanation)
- **Matching Percentage:** (0-100%)

Format each candidate like this:
1. **Name**
   - **Relevant Skills:** skill1, skill2, skill3
   - **Why They Are a Good Fit:** explanation
   - **Matching Percentage:** XX%

Answer:"""

    llm = ChatOpenAI(model="gpt-4o-mini", temperature=0)
    response = llm.invoke(prompt)

    print("\n" + "="*60)
    print("🎯 SEARCH RESULTS")
    print("="*60)
    print(response.content)
    print("="*60)

    return response.content

def clear_resumes():
    """Clear all resumes from vector database"""
    print("🗑️  Clearing resume database...")
    if os.path.exists(Config.DB_DIR):
        shutil.rmtree(Config.DB_DIR)
        os.makedirs(Config.DB_DIR, exist_ok=True)
        return "Database cleared successfully"
    else:
        return "No database found"

def count_resumes():
    """Count resume files waiting to be ingested"""
    if not os.path.exists(Config.RESUMES_DIR):
        return "0 resumes found - folder doesn't exist"
    files = [f for f in os.listdir(Config.RESUMES_DIR) if f.endswith(('.txt', '.pdf'))]
    return f"Found {len(files)} resume files: {', '.join(files)}"

# ============================================================================
# LANGCHAIN TOOLS
# ============================================================================

@tool
def ingest_resumes_tool():
    """Ingest new resumes from the './resumes' folder into the vector database. Use this tool when new resumes need to be processed or the database needs to be updated."""
    return ingest_resumes()

@tool
def list_resumes_tool():
    """List all the unique resume file names currently stored in the vector database. Use this tool to see what resumes have been ingested."""
    return list_resumes()

@tool
def search_resumes_tool(skills: str):
    """Search for candidates whose resumes match the given skills. Input should be a comma-separated string of required skills (e.g., 'Python, Machine Learning, Docker'). Use this tool to find candidates for a job opening."""
    return search_resumes(skills)

@tool
def clear_resumes_tool():
    """Clear all resumes from the vector database. This will delete the entire resume database. Use this tool to start fresh or remove all stored resume data."""
    return clear_resumes()

@tool
def count_resumes_tool():
    """Count how many resume files are waiting in the './resumes' folder to be ingested. Use this to check if there are new resumes to process."""
    return count_resumes()

# ============================================================================
# SIMPLE AGENTIC SYSTEM (Using OpenAI Function Calling)
# ============================================================================

class ResumeAgent:
    def __init__(self):
        self.llm = ChatOpenAI(model="gpt-4o-mini", temperature=0)

        self.tools = [
            ingest_resumes_tool,
            list_resumes_tool,
            search_resumes_tool,
            clear_resumes_tool,
            count_resumes_tool
        ]

        # Bind tools to LLM (OpenAI function calling)
        self.llm_with_tools = self.llm.bind_tools(self.tools)

        self.system_message = """You are an AI assistant helping with resume management. You have access to tools to manage and search resumes.

When a user asks a question:
1. Think about which tool(s) you need to use
```

```python
2. Use the appropriate tool(s) to get the information
3. Provide a clear, helpful response based on the tool results

Available tools:
- ingest_resumes_tool: Process new resumes from the folder
- list_resumes_tool: Show all resumes in the database
- search_resumes_tool: Find candidates matching specific skills
- clear_resumes_tool: Delete all resumes from database
- count_resumes_tool: Count resumes waiting to be processed

Be helpful, concise, and accurate."""

    def run(self, query: str, max_iterations: int = 5):
        """Run the agent with a query"""
        try:
            messages = [
                SystemMessage(content=self.system_message),
                HumanMessage(content=query)
            ]

            steps = []

            for iteration in range(max_iterations):
                print(f"\n🔄 Iteration {iteration + 1}")

                # Get LLM response with tool calls
                response = self.llm_with_tools.invoke(messages)
                messages.append(response)

                # Check if there are tool calls
                if not response.tool_calls:
                    # No more tool calls, return final answer
                    return {
                        "output": response.content,
                        "steps": steps
                    }

                # Execute tool calls
                for tool_call in response.tool_calls:
                    tool_name = tool_call["name"]
                    tool_args = tool_call["args"]

                    print(f"🔧 Calling tool: {tool_name} with args: {tool_args}")

                    # Find and execute the tool
                    tool_func = None
                    for tool in self.tools:
                        if tool.name == tool_name:
                            tool_func = tool
                            break

                    if tool_func:
                        try:
                            # Execute tool
                            if tool_args:
                                result = tool_func.invoke(tool_args)
                            else:
                                result = tool_func.invoke({})

                            print(f"✓ Tool result: {str(result)[:200]}...")

                            # Add tool result to messages
                            messages.append(
                                ToolMessage(
                                    content=str(result),
                                    tool_call_id=tool_call["id"]
                                )
                            )

                            steps.append({
                                "tool": tool_name,
                                "input": str(tool_args),
                                "output": str(result)[:200]
                            })
                        except Exception as e:
                            error_msg = f"Error executing {tool_name}: {str(e)}"
                            print(f"❌ {error_msg}")
                            messages.append(
                                ToolMessage(
                                    content=error_msg,
                                    tool_call_id=tool_call["id"]
                                )
                            )
                    else:
                        print(f"❌ Tool {tool_name} not found")

            # Max iterations reached
            return {
                "output": "Maximum iterations reached. Please try rephrasing your question.",
                "steps": steps
            }

        except Exception as e:
            print(f"❌ Agent execution error: {str(e)}")
            return {
                "output": f"Error: {str(e)}",
                "steps": []
            }

# ============================================================================
# FASTAPI APP
# ============================================================================

app = FastAPI(
    title="Agentic Resume Management API",
    description="LangChain Agent-powered resume management with autonomous decision-making",
    version="3.0.0"
)

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

if os.path.exists("static"):
    app.mount("/static", StaticFiles(directory="static"), name="static")

# Global agent instance
agent = None

# ============================================================================
# MODELS
# ============================================================================

class AgentRequest(BaseModel):
    query: str

class AgentResponse(BaseModel):
    query: str
    response: str
    agent_steps: Optional[List[dict]] = None
    timestamp: str

class SearchRequest(BaseModel):
    skills: str

class UploadResponse(BaseModel):
    status: str
    message: str
    filename: str
    timestamp: str

# ============================================================================
# ENDPOINTS
# ============================================================================

@app.get("/")
async def root():
    return {
        "message": "Agentic Resume Management API",
        "version": "3.0.0",
        "mode": "OPENAI FUNCTION CALLING AGENT",
        "web_interface": "/static/index.html",
        "endpoints": {
            "agent": "/agent (POST) - Natural language queries",
            "search": "/search (POST) - Direct search",
            "upload": "/upload (POST) - Upload resume",
            "health": "/health",
            "docs": "/docs"
        }
    }

@app.post("/agent", response_model=AgentResponse, tags=["Agent"])
async def agent_query(request: AgentRequest):
    """
    🤖 AGENTIC ENDPOINT - Send natural language queries

    Examples:
    - "What resumes do I have?"
    - "Find candidates who know React and Node.js"
    - "Ingest all new resumes"
    - "How many resumes are waiting to be processed?"
    """
    try:
        if not request.query or not request.query.strip():
            raise HTTPException(status_code=400, detail="Query cannot be empty")

        print(f"\n{'='*60}")
        print(f"🤖 AGENT QUERY: {request.query}")
        print(f"{'='*60}\n")

        # Run agent
        result = agent.run(request.query)

        return AgentResponse(
            query=request.query,
            response=result["output"],
            agent_steps=result.get("steps"),
            timestamp=datetime.now().isoformat()
        )

    except Exception as e:
        print(f"❌ Agent error: {str(e)}")
        raise HTTPException(status_code=500, detail=f"Agent error: {str(e)}")

@app.post("/search", tags=["Direct"])
async def search_resumes_endpoint(request: SearchRequest):
    """Direct search endpoint (non-agentic fallback)"""
    try:
        if not request.skills:
            raise HTTPException(status_code=400, detail="Skills cannot be empty")

        results = search_resumes(request.skills)

        return {
            "query": request.skills,
            "results": results,
            "timestamp": datetime.now().isoformat()
        }
    except Exception as e:
        raise HTTPException(status_code=500, detail=str(e))

@app.post("/upload", response_model=UploadResponse, tags=["Upload"])
async def upload_resume(file: UploadFile = File(...)):
    """Upload a resume file"""
    try:
        if not file.filename:
            raise HTTPException(status_code=400, detail="No filename")

        ext = os.path.splitext(file.filename)[1].lower()
        if ext not in ['.txt', '.pdf']:
            raise HTTPException(status_code=400, detail="Only .txt and .pdf allowed")

        os.makedirs(Config.RESUMES_DIR, exist_ok=True)

        safe_filename = file.filename.replace(" ", "_")
        file_path = os.path.join(Config.RESUMES_DIR, safe_filename)

        if os.path.exists(file_path):
            raise HTTPException(status_code=409, detail=f"File '{safe_filename}' exists")

        async with aiofiles.open(file_path, 'wb') as f:
            content = await file.read()
            await f.write(content)

        return UploadResponse(
            status="success",
            message=f"Uploaded successfully. Use agent query: 'Ingest new resumes'",
            filename=safe_filename,
            timestamp=datetime.now().isoformat()
        )
    except HTTPException:
        raise
    except Exception as e:
        raise HTTPException(status_code=500, detail=str(e))

@app.get("/health", tags=["System"])
async def health():
    """Health check"""
    return {
        "status": "healthy",
        "mode": "OPENAI FUNCTION CALLING",
        "agent_type": "Custom Agent with OpenAI Function Calling",
        "api_version": "3.0.0",
        "tools_count": len(agent.tools) if agent else 0
    }

@app.on_event("startup")
async def startup_event():
    """Initialize on startup"""
    global agent

    print("\n" + "="*60)
    print("🤖 OPENAI FUNCTION CALLING AGENT Starting...")
    print("="*60)

    Config.setup()
    agent = ResumeAgent()

    print(f"✓ Agent Mode: OPENAI FUNCTION CALLING")
    print(f"✓ Tools: {len(agent.tools)}")
    print(f"✓ LLM: gpt-4o-mini")
    print(f"✓ API Key: {os.getenv('OPENAI_API_KEY')[:20]}...")
    print("\n🌐 Web Interface: http://localhost:8000/static/index.html")
    print("🤖 Agent Endpoint: http://localhost:8000/agent")
    print("📚 API Docs: http://localhost:8000/docs")
    print("="*60 + "\n")

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