1. Setup and Initialization

```
import pandas as pd
from docx import Document # For DOCX parsing
from sklearn.metrics.pairwise import cosine_similarity
# Placeholder for LLM/Embedding libraries
# If using OpenAI:
# from openai import OpenAI
# If using Hugging Face (local/open source):
# from transformers import AutoModel, AutoTokenizer
# import torch
# import faiss # For local Vector DB

# --- Configuration Placeholder ---
# API_KEY = "YOUR_API_KEY"
# EMBEDDING_MODEL_NAME = "text-embedding-3-small" # e.g., OpenAI
# LLM_MODEL_NAME = "gpt-4-turbo" # e.g., OpenAI

# Placeholder for the Vector Database
class VectorDB:
    def __init__(self):
        self.index = []
        self.texts = []

    def add_document(self, text_chunks, embeddings):
        """Stores text chunks and their corresponding embeddings."""
        self.texts.extend(text_chunks)
        self.index.extend(embeddings)

    def search(self, query_embedding, k=3):
        """Performs cosine similarity search to find top-K chunks."""
        if not self.index:
            return []

        # Calculate similarity (simplified concept - replace with Faiss/ChromaDB)
        similarities = cosine_similarity([query_embedding], self.index)[0]
        top_k_indices = similarities.argsort()[-k:][::-1]

        results = []
        for i in top_k_indices:
            results.append({
                "text": self.texts[i],
                "score": similarities[i]
            })
        return results

vector_db = VectorDB()
```

2. Data Ingestion, Chunking, and Embedding

```
# --- 1. Data Ingestion and Parsing ---1
def extract_text_from_docx(file_path):
    """Simple extraction from DOCX file."""
    doc = Document(file_path)
    text = ""
    for para in doc.paragraphs:
        text += para.text + "\n"
    return text

def chunk_text(text, chunk_size=500, overlap=100):
    """Simple text chunking based on character count."""
    chunks = []
    # A real implementation would use a more sophisticated splitter (e.g., LangChain's RecursiveTextSplitter)
    i = 0
    while i < len(text):
        chunks.append(text[i:i + chunk_size])
        i += chunk_size - overlap
    return chunks

# --- 2. Embedding (Placeholder) ---
def generate_embedding(text):
    """
    Placeholder function for generating embeddings using an external model.
    In reality, this calls the OpenAI API or a local HuggingFace model.
    """
    # Dummy embedding: a list of numbers representing the text
    # In a real project, use your chosen embedding model (e.g., OpenAI, Sentence-Transformer)
    import hashlib
    # Creates a stable-ish 128-dimensional vector based on the hash of the text
    hash_value = hashlib.sha256(text.encode()).hexdigest()
    # The hash_value is 64 characters long (256 bits / 4 bits per hex char)
    # We need to iterate only up to the length of the hash_value.
    vector = [int(hash_value[i:i+2], 16) / 255.0 for i in range(0, len(hash_value), 2)]
    return vector[:128] # Truncate to 128 dimensions

# --- Pipeline Execution ---
def index_resume(file_path):
    """Indexes a single resume into the Vector DB."""
    resume_text = extract_text_from_docx(file_path)
    text_chunks = chunk_text(resume_text)

    embeddings = [generate_embedding(chunk) for chunk in text_chunks]

    vector_db.add_document(text_chunks, embeddings)
    print(f"Indexed {len(text_chunks)} chunks from the resume.")

# Use the file content from the user's uploaded resume for the demo
RESUME_PATH = "Shreyas_Rastogi_Data_Scientist_AI_Resume_UPDATED_SKILLS.docx"
index_resume(RESUME_PATH)
```

```
Indexed 8 chunks from the resume.
```

3. Retrieval-Augmented Generation (RAG)

```
# --- 4. LLM Evaluation and Output Generation (Placeholder) ---
def get_llm_response(prompt):
    """
    Placeholder for the final LLM call (e.g., OpenAI or Llama 3 API).
    """
    print("\n--- LLM is processing the prompt (context + JD) ---")

    # Simple, hardcoded evaluation for demonstration, based on Shreyas's resume.
    if "Data Scientist" in prompt and "Credit Card Fraud Detection" in prompt:
        return {
            "match_score": 90,
            "fit_summary": "Excellent fit for a Data Scientist role with a focus on AI/ML. Strong practical experience in de
            "top_strengths": ["Deep Learning/NLP (BERT, Transformers) [cite: 25][cite_start]", "MLOps Pipeline Development [
            "gaps": ["Lacks explicit mention of leadership experience or large-scale production deployment history (typical
        }
    return {"match_score": 50, "fit_summary": "General match."}


# --- 3. Resume Screening and Retrieval ---
def screen_resume(job_description, k=5):
    """
    Performs the RAG process:
    1. Embeds the JD.
    2. Retrieves relevant resume chunks.
    3. Prompts the LLM with the full context.
    """

    # 3.1. Embed the Job Description (Query)
    jd_embedding = generate_embedding(job_description)

    # 3.2. Retrieval (RAG)
    retrieved_chunks = vector_db.search(jd_embedding, k=k)

    # Compile the retrieved context for the LLM
    context_text = "\n---\n".join([item['text'] for item in retrieved_chunks])

    # 3.3. Prompt Engineering
    system_prompt = (
        "You are an expert technical recruiter analyzing a candidate's resume against a Job Description (JD). "
        "Your task is to provide a match assessment based ONLY on the provided CONTEXT (the resume snippets)."
        "Output the result as a JSON object with 'match_score' (0-100), 'fit_summary', 'top_strengths', and 'gaps'."
    )

    user_prompt = f"""
    ### JOB DESCRIPTION ###
    {job_description}

    ### CONTEXT (Relevant Resume Snippets) ###
    {context_text}

    ### ASSESSMENT ###
    Generate the JSON assessment.
    """
    # 3.4. LLM Evaluation
    llm_assessment = get_llm_response(user_prompt)

    return llm_assessment

# --- Example Usage (Simulated) ---
job_description = """
Data Scientist - AI/ML Engineer:
Seeking a skilled Data Scientist with expertise in Deep Learning, NLP (Transformers), and MLOps.
Must have experience in building end-to-end pipelines and dealing with imbalanced data/fraud detection models.
Required Skills: Python, TensorFlow, Transformers, AWS, Flask.
"""

# Run the full screening process
assessment_result = screen_resume(job_description)

## Output the result
print("\n" + "="*50)
print("🎯 LLM RESUME SCREENING RESULT 🎯")
print("="*50)
print(f"Match Score: {assessment_result['match_score']}/100")
print(f"Summary: {assessment_result['fit_summary']}")
print("\nTop Strengths:")
for strength in assessment_result['top_strengths']:
    print(f"- {strength}")
print("\nGaps:")
print(f"- {assessment_result['gaps'][0]}")
print("="*50)
```

```
--- LLM is processing the prompt (context + JD) ---

==================================================
🎯 LLM RESUME SCREENING RESULT 🎯
==================================================
Match Score: 90/100
Summary: Excellent fit for a Data Scientist role with a focus on AI/ML. Strong practical experience in deep learning (Autoer

Top Strengths:
- Deep Learning/NLP (BERT, Transformers) [cite: 25][cite_start]
- MLOps Pipeline Development [cite: 45][cite_start]
- Unsupervised Anomaly Detection [cite: 20]

Gaps:
- Lacks explicit mention of leadership experience or large-scale production deployment history (typical for a B.Tech graduat
==================================================
```