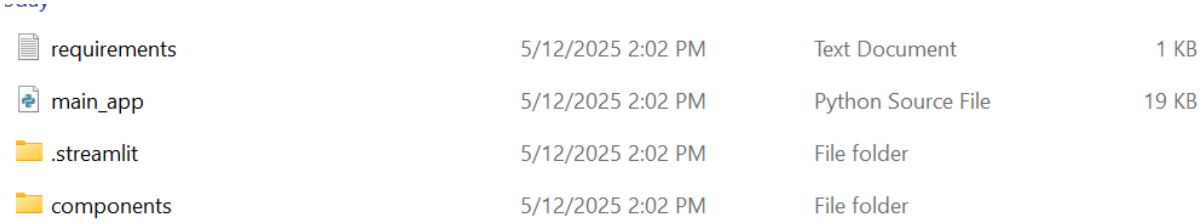
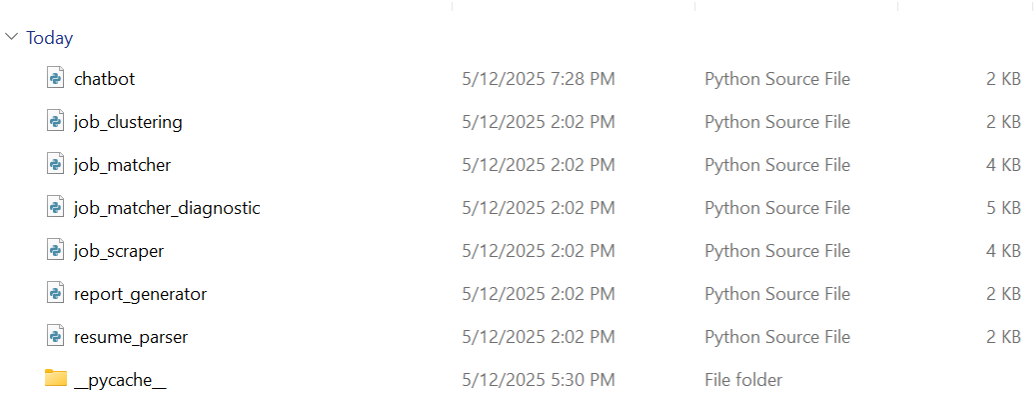
File Structure:

A folder named ‘Resume_Application_Final’ contains the following contents.

- ➔ The ‘main_app.py’ is our main file which contains the code for the application
- ➔ ‘requirements.txt’ consists of all the libraries and modules, along with their version number, that need to be run before executing the main code. Use the following code to install the appropriate modules “pip install -r requirements.txt”



- ➔ The folder named ‘component’ has the code for all the other functionalities used in the main application. See below.



***We will explore all the components in the relevant order first and then understand the main code file**

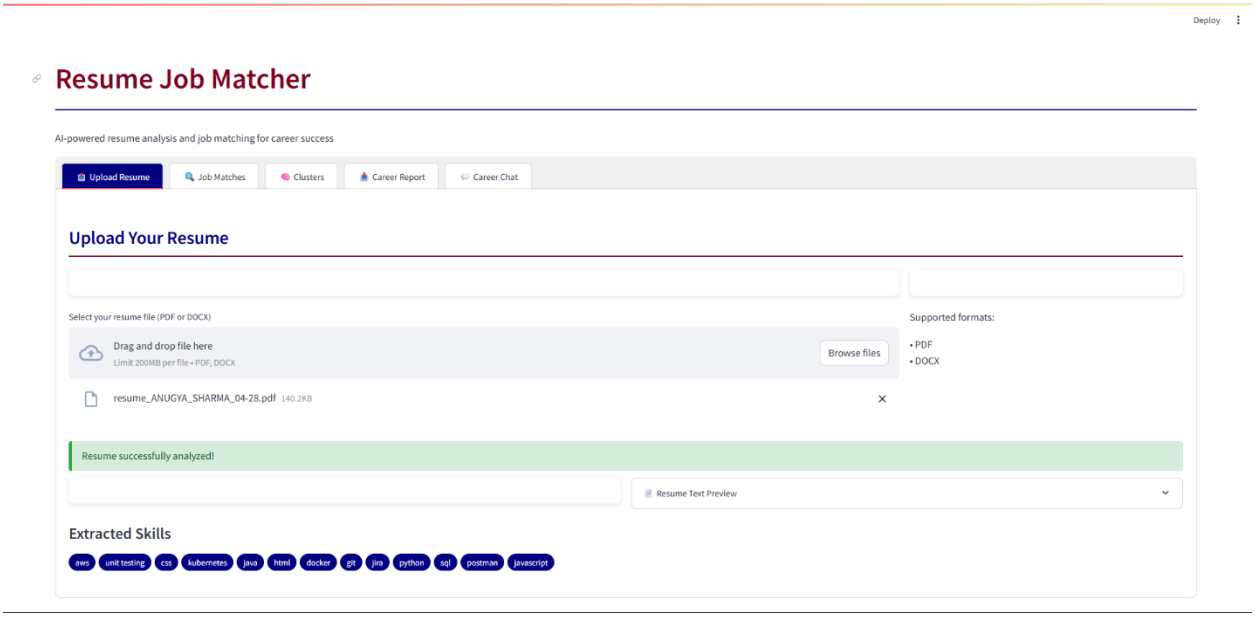
1. resume_parser.py:

(Extracts raw text and relevant technical skills from uploaded resumes (PDF/DOCX).)

Code:

```
import docx2txt
import PyPDF2
import re
import string

# --- Unified Skill Set ---
SKILL_KEYWORDS = list(set([
    "java", "python", "sql", "aws", "azure", "html", "css", "javascript", "react", "nodejs", "tensorflow",
    "keras", "pandas", "numpy", "scikit-learn", "git", "docker", "kubernetes", "flask", "django", "spark",
    "hadoop", "nlp", "c++", "c#", "xgboost", "airflow", "jira", "postman", "ci/cd", "unit testing", "linux"
]))

# --- Text Extraction ---
def extract_text(file):
    if file.type == "application/pdf":
        reader = PyPDF2.PdfReader(file)  #Reads PDF uploads
        return " ".join([page.extract_text() or "" for page in reader.pages])
    elif file.type == "application/vnd.openxmlformats-officedocument.wordprocessingml.document":
        return docx2txt.process(file) #Reads txt uploads
    return ""

# --- Skill Extraction ---
def extract_skills(text, keywords=SKILL_KEYWORDS):
    text = text.lower() #Converts all text to lowercase
    text = text.translate(str.maketrans("", "", string.punctuation)) # Removes all punctuation characters
from the text.
    text = re.sub(r"\s+", " ", text)   # Replaces multiple whitespace characters with a single space
    return list({skill for skill in keywords if skill.lower() in text})

# --- Entry Point ---
def parse_resume_and_skills(file):
    text = extract_text(file)
    skills = extract_skills(text)
    return text, skills
```

Output:



2. job_scraper.py

(Fetches job listings via RapidAPI based on the extracted resume skills.)

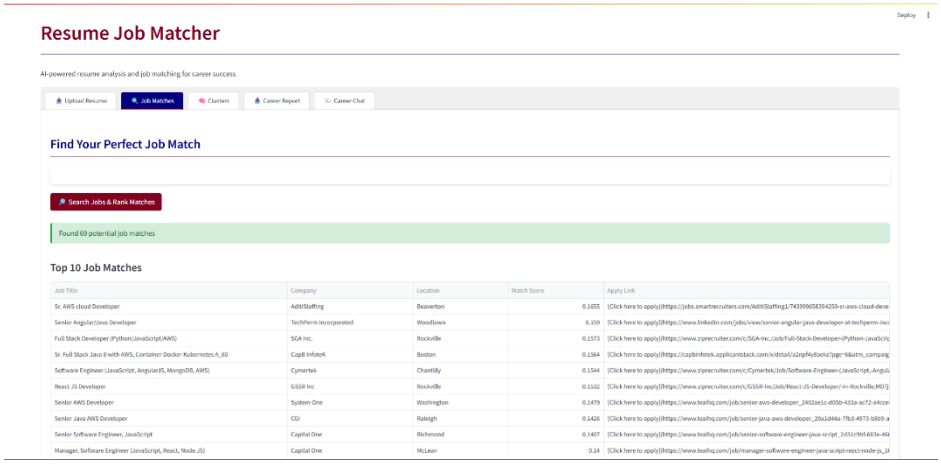
Code:

```
import requests
import random
import pandas as pd
import traceback
import time

# --- Live Job Scraper ---
def get_all_jobs(skills):
    """
    Fetch jobs based on skills from RapidAPI JSSearch
    With added debugging information
    """
    print(f"Starting job search with skills: {skills}")

    if not skills or len(skills) == 0:
        print("Error: No skills provided for job search")
        return pd.DataFrame()
```

```
headers = {
    "X-RapidAPI-Key": "36e3982643msh28252de82063dc1p1779cbjsn060c60c0604f",
    "X-RapidAPI-Host": "jsearch.p.rapidapi.com"
}
url = "https://jsearch.p.rapidapi.com/search"

# Create search queries from skills (pairs of skills)
queries = [" ".join(skills[i:i+2]) for i in range(0, len(skills), 2)]
if not queries:
    # Fallback to individual skills if pairing doesn't work
    queries = skills

print(f"Generated {len(queries)} search queries: {queries}")

all_jobs = []
seen = set()

# Take a random sample of queries to avoid excessive API calls
sample_size = min(len(queries), 5)
query_sample = random.sample(queries, sample_size)
print(f"Sampling {sample_size} queries: {query_sample}")

for q in query_sample:
    print(f"\nSearching for jobs with query: '{q}'")
    for page in range(1, 3):  # Check first 2 pages
        print(f"  Checking page {page}...")
        params = {"query": q, "page": str(page), "num_pages": "1"}

        try:
            # Add delay to avoid rate limiting
            time.sleep(0.5)

            # Make the API request
            resp = requests.get(url, headers=headers, params=params)

            # Check response status
            if resp.status_code != 200:
                print(f"  Error: API returned status code {resp.status_code}")
                print(f"  Response: {resp.text[:200]}...")
                continue

            # Parse response
            response_json = resp.json()
            jobs = response_json.get("data", [])

            if not jobs:
                print(f"  No jobs found on page {page}")
                continue
```

```
            print(f"  Found {len(jobs)} jobs on page {page}")

            # Process jobs
            for job in jobs:
                link = job.get("job_apply_link")
                if not link or link in seen:
                    continue

                seen.add(link)
                all_jobs.append({
                    "Skill": q,
                    "Job Title": job.get("job_title"),
                    "Company": job.get("employer_name"),
                    "Location": job.get("job_city"),
                    "Apply Link": f"[Click here to apply]({link})",
                    "Description": job.get("job_description", "")
                })

        except Exception as e:
            print(f"  Error processing query '{q}' page {page}:")
            print(f"  {str(e)}")
            print(traceback.format_exc())
            continue

print(f"\nTotal jobs collected: {len(all_jobs)}")

# Return DataFrame or empty DataFrame if no jobs found
if all_jobs:
    return pd.DataFrame(all_jobs)
else:
    print("No jobs found across all queries")
    return pd.DataFrame()
```

Output:



3. Job_matcher.py:

(Ranks jobs by computing semantic similarity between resume content and job descriptions using sentence embeddings.)

Code:

```
import os
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity
import pandas as pd
import re

os.environ["TF_ENABLE_ONEDNN_OPTS"] = "0"

# --- Match and Score Jobs ---
def rank_jobs_by_embedding(resume_text, job_df, skill_list):
    """
    Rank jobs by similarity to resume text with enhanced debugging
    """
    print(f"\n--- Starting job ranking process ---")
    print(f"Resume text length: {len(resume_text)} characters")
    print(f"Skills list: {skill_list}")

    # Check if job dataframe is empty
    if job_df is None or job_df.empty:
        print("Error: Job dataframe is empty or None")
        return pd.DataFrame(), []

    print(f"Job dataframe shape: {job_df.shape}")
    print(f"Job dataframe columns: {job_df.columns.tolist()}")

    # Check if Description column exists
    if 'Description' not in job_df.columns:
        print("Warning: 'Description' column not found in job dataframe")
        print("Available columns:", job_df.columns.tolist())

        # Try to find an alternative description column
        desc_alternatives = ['description', 'job_description', 'JobDescription']
        found = False
        for col in desc_alternatives:
            if col in job_df.columns:
                print(f"Using '{col}' instead of 'Description'")
                job_df['Description'] = job_df[col]
                found = True
                break
```

```
    if not found:
        # Create a placeholder description from job title
        print("Creating placeholder descriptions from job titles")
        job_df['Description'] = job_df.apply(
            lambda row: f"{row.get('Job Title', '')} at {row.get('Company', '')}",
            axis=1
        )

# Check for any empty descriptions
empty_desc_count = job_df['Description'].isnull().sum()
if empty_desc_count > 0:
    print(f"Warning: {empty_desc_count} jobs have empty descriptions")
    job_df = job_df.dropna(subset=['Description'])
    print(f"After dropping NaN descriptions: {len(job_df)} jobs remain")

# Create document collection for TF-IDF
print("Creating TF-IDF vectorizer...")
docs = [resume_text] + job_df['Description'].tolist()

# Create and apply TF-IDF vectorizer
try:
    vec = TfidfVectorizer(stop_words='english')
    vectors = vec.fit_transform(docs)
    print(f"TF-IDF vectorization successful. Matrix shape: {vectors.shape}")

    # Calculate similarity scores
    scores = cosine_similarity(vectors[0:1], vectors[1:]).flatten()
    print(f"Calculated similarity scores. Range: {scores.min():.4f} to {scores.max():.4f}")

    # Add scores to dataframe and sort
    job_df['Match Score'] = scores
    job_df = job_df.sort_values(by='Match Score', ascending=False)

except Exception as e:
    print(f"Error during vectorization or similarity calculation: {str(e)}")
    return job_df, []

# Find missing skills
print("Analyzing skill gaps...")
clean_text = re.sub(r"[\W_]+", " ", resume_text.lower())
resume_skills = [skill for skill in skill_list if skill.lower() in clean_text]

print(f"Skills found in resume: {len(resume_skills)}/{len(skill_list)}")
print(f"Found skills: {resume_skills}")

# Identify missing skills that appear in top jobs
top_descriptions = " ".join(job_df['Description'].head(5)).lower()
missing = [s for s in skill_list if s.lower() not in resume_skills and s.lower() in top_descriptions]
```

```
    print(f"Skills missing from resume but present in top jobs: {missing}")
    print("--- Job ranking complete ---")
    return job_df, missing
```

Output:
Refer to the ‘Match Score’ column in (2) above

4. Job_matcher_diagnostic.py

(This script tests the functionality of job scraping and job matching modules using sample skills and a resume, and reports any issues or successes.)

Code:

```
import pandas as pd
import sys
import traceback

# Test script to diagnose job scraping and matching issues
print("===== JOB MATCHER DIAGNOSTICS =====")

# Import the debugging versions - replace with your actual imports
try:
    # First, try to import your actual modules
    try:
        print("Trying to import original modules...")
        from components.job_scraper import get_all_jobs
        from components.job_matcher import rank_jobs_by_embedding
    except ImportError:
        print("Couldn't import from components, trying direct imports...")
        # If that fails, import directly from current directory
        from job_scraper import get_all_jobs
        from job_matcher import rank_jobs_by_embedding

    print("Modules imported successfully")
except Exception as e:
    print(f"Import error: {str(e)}")
    print(traceback.format_exc())
    sys.exit(1)

# Test data
print("\n1. SETTING UP TEST DATA")
test_skills = ["python", "data analysis", "machine learning", "sql", "javascript"]
print(f"Test skills: {test_skills}")

test_resume = """
John Doe
Data Scientist
```

```
Skills:
- Python programming
- Data analysis and visualization
- Machine learning models
- SQL database management
- Project management

Experience:
- Data Scientist at XYZ Corp (2020-Present)
- Data Analyst at ABC Inc (2018-2020)
"""
print(f"Test resume length: {len(test_resume)} characters")

# Test job scraping
print("\n2. TESTING JOB SCRAPER")
try:
    print("Calling get_all_jobs()...")
    jobs_df = get_all_jobs(test_skills)

    if jobs_df is None:
        print("ERROR: get_all_jobs returned None")
    elif jobs_df.empty:
        print("WARNING: get_all_jobs returned empty DataFrame")
    else:
        print(f"SUCCESS: Found {len(jobs_df)} jobs")
        print(f"DataFrame columns: {jobs_df.columns.tolist()}")
        print("\nFirst job sample:")
        if len(jobs_df) > 0:
            sample_job = jobs_df.iloc[0]
            for col in jobs_df.columns:
                val = sample_job[col]
                if col == 'Description':
                    val = val[:100] + "..." if isinstance(val, str) and len(val) > 100 else val
                print(f"  {col}: {val}")
except Exception as e:
    print(f"Job scraper error: {str(e)}")
    print(traceback.format_exc())
    # Continue to next test rather than exiting

# Test job matching
print("\n3. TESTING JOB MATCHER")
try:
    # Create a fake jobs dataframe if the real one failed
    if 'jobs_df' not in locals() or jobs_df is None or jobs_df.empty:
        print("Using test data for job matching since scraper failed")
        jobs_df = pd.DataFrame({
            'Job Title': ['Data Scientist', 'Data Analyst', 'ML Engineer'],
            'Company': ['Test Co', 'Sample Inc', 'Demo LLC'],
```

```
            'Location': ['Remote', 'New York', 'San Francisco'],
            'Description': [
                'Looking for a Python expert with machine learning skills',
                'Need SQL and data analysis expert',
                'AI/ML engineer with deep learning experience'
            ]
        })

    print(f"Running job matching with {len(jobs_df)} jobs...")
    ranked_df, missing = rank_jobs_by_embedding(test_resume, jobs_df, test_skills)

    if ranked_df is None:
        print("ERROR: rank_jobs_by_embedding returned None for ranked_df")
    elif ranked_df.empty:
        print("WARNING: rank_jobs_by_embedding returned empty DataFrame")
    else:
        print(f"SUCCESS: Ranked {len(ranked_df)} jobs")
        print(f"Missing skills: {missing}")
        print("\nTop 3 matches:")

        # Show top 3 matches
        top_matches = ranked_df.head(3)
        for i, (_, job) in enumerate(top_matches.iterrows()):
            print(f"\nMatch #{i+1}: {job.get('Job Title', 'Untitled')} at {job.get('Company', 'Unknown')}")
            print(f"  Match Score: {job.get('Match Score', 'N/A'):.4f}")

except Exception as e:
    print(f"Job matcher error: {str(e)}")
    print(traceback.format_exc())

print("\n===== DIAGNOSTICS COMPLETE =====")
```

5. job_clustering.py

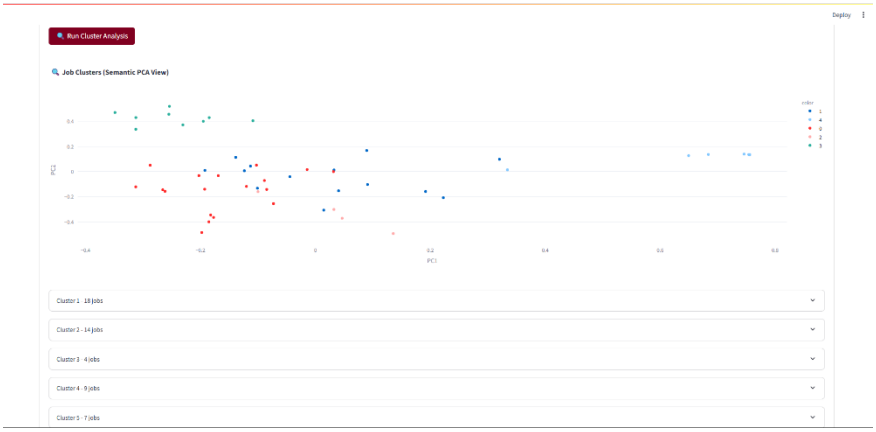
(Clusters the ranked job matches into similar career paths using KMeans clustering and visualizes them.)

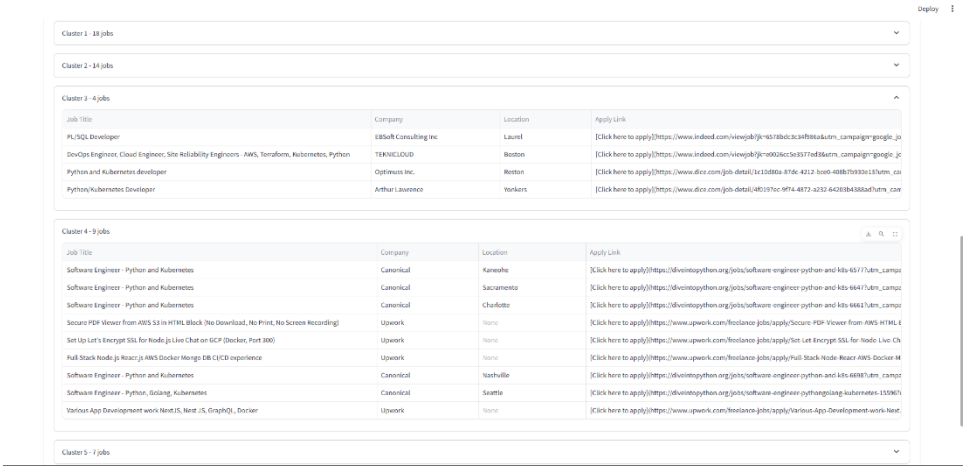
Code:

```
import pandas as pd
from sklearn.cluster import KMeans
from sklearn.decomposition import PCA
import plotly.express as px
from sentence_transformers import SentenceTransformer

# --- Cluster Jobs ---
def cluster_jobs(jobs_df, num_clusters=3):
    if jobs_df.empty or len(jobs_df) < num_clusters:
        return jobs_df, None

    if jobs_df['Description'].isnull().all():
```

```
        raise ValueError("All job descriptions are empty. Cannot perform clustering.")

    model = SentenceTransformer("all-MiniLM-L6-v2")
    X = model.encode(jobs_df['Description'].fillna(""), show_progress_bar=False)
    kmeans = KMeans(n_clusters=num_clusters, random_state=42, n_init=10)
    labels = kmeans.fit_predict(X)
    jobs_df['Cluster'] = labels

    return jobs_df, visualize_clusters(jobs_df, X, labels)

# --- Plotly Visualization ---
def visualize_clusters(jobs_df, features, labels):
    pca = PCA(n_components=2)
    reduced = pca.fit_transform(features)
    df_plot = pd.DataFrame(reduced, columns=['PC1', 'PC2'])
    df_plot['Cluster'] = labels
    df_plot['Job Title'] = jobs_df['Job Title']
    df_plot['Company'] = jobs_df['Company']

    fig = px.scatter(
        df_plot,
        x='PC1', y='PC2',
        color=df_plot['Cluster'].astype(str),
        hover_data=['Job Title', 'Company'],
        title="🔍 Job Clusters (Semantic PCA View)"
    )
    fig.update_layout(height=500)
    return fig
```

Output:





6. report_generator.py

(Generates a styled HTML report summarizing the user's resume insights, job matches, and cluster analysis.)

Code:

```
import pandas as pd
import tempfile
from jinja2 import Template

# --- Generate HTML Report ---
def generate_report(resume_text, skills, ranked_df, clustered_df):
    top_jobs = ranked_df[['Job Title', 'Company', 'Location', 'Match Score']].head(10).to_html(index=False)

    clusters = []
    for cluster_id in sorted(clustered_df['Cluster'].unique()):
        group = clustered_df[clustered_df['Cluster'] == cluster_id]
```

```
    html_table = group[['Job Title', 'Company', 'Location', 'Match Score']].head(5).to_html(index=False)
    clusters.append({"label": f"Cluster {cluster_id + 1}", "table": html_table})

# HTML Template
html_template = Template("""
<html>
<head>
    <style>
        body { font-family: 'Arial', sans-serif; padding: 20px; }
        h1, h2, h3 { color: #2c3e50; }
        table { width: 100%; border-collapse: collapse; margin-bottom: 20px; }
        th, td { border: 1px solid #ccc; padding: 8px; }
        th { background-color: #f2f2f2; }
    </style>
</head>
<body>
    <h1>Career Report</h1>
    <h2>Extracted Skills</h2>
    <p>{{ skills }}</p>
    <h2>Top Job Matches</h2>
    {{ top_jobs|safe }}
    <h2>Clustered Recommendations</h2>
    {% for c in clusters %}
        <h3>{{ c.label }}</h3>
        {{ c.table|safe }}
    {% endfor %}
</body>
</html>
""")

html_out = html_template.render(skills=", ".join(skills), top_jobs=top_jobs, clusters=clusters)
tmp_file = tempfile.NamedTemporaryFile(delete=False, suffix=".html")
with open(tmp_file.name, "w", encoding="utf-8") as f:
    f.write(html_out)
return tmp_file.name
```

Output:

Career Report

Extracted Skills

java, css, kubernetes, git, sql, postman, docker, javascript, aws, html, jira, python, unit testing

Top Job Matches

Job Title	Company	Location	Match Score
Java Fullstack Developer	CSS Tec Staffing	Conshohocken	0.217176
Full Stack Java Software Engineer	CSS Tec Staffing	Wilmington	0.189538
NodeJS Developer - Nodejs, Java, JavaScript, DevOps, Kubernetes, AWS/Azure/GCP	IBM	None	0.168253
Java Fullstack Engineer	I20 Technologies	Plano	0.156532
Java Engineer with Docker and Kubernetes	Darbas, LLC	San Francisco	0.140295
Software Engineer - Golang / Kubernetes / DevOps	Request Technology	Chicago	0.139301
Senior Drupal Developer with AWS and backend scripting experience	The Consortium, Inc.	Rockville	0.136676
Java Full Stack Developer	Deloitte	Honolulu	0.136205
Junior UI/UX Developer (HTML, CSS, JavaScript) Job at SynergisticIT in St Louis	SynergisticIT	St. Louis	0.130054
PHP Web Developer	Cynet Systems	St. Louis	0.129899

Clustered Recommendations

Cluster 1

Job Title	Company	Location	Match Score
Java Fullstack Developer	CSS Tec Staffing	Conshohocken	0.217176
Java Fullstack Engineer	I20 Technologies	Plano	0.156532
Junior UI/UX Developer (HTML, CSS, JavaScript) Job at SynergisticIT in St Louis	SynergisticIT	St. Louis	0.130054
PHP Web Developer	Cynet Systems	St. Louis	0.129899
Full Stack Software Engineer	PerformYard	Dothan	0.124363

Cluster 2

Job Title	Company	Location	Match Score
NodeJS Developer - Nodejs, Java, JavaScript, DevOps, Kubernetes, AWS/Azure/GCP	IBM	None	0.168253
Junior and Senior Front-End Java Developer	Collabera	None	0.118573
Groovy Developer	Diverse Linx	McLean	0.082364
SWE 2 - Kubernetes, Linux - to 200k+ - FS Poly	Scientific Research Corporation	Annapolis Junction	0.078545
JIRA Developer - Dallas, TX	Info-Ways	Dallas	0.069320

Cluster 3

Job Title	Company	Location	Match Score
Full Stack Java Software Engineer	CSS Tec Staffing	Wilmington	0.189538
Java Engineer with Docker and Kubernetes	Darbas, LLC	San Francisco	0.140295
Software Engineer - Golang / Kubernetes / DevOps	Request Technology	Chicago	0.139301
Senior Drupal Developer with AWS and backend scripting experience	The Consortium, Inc.	Rockville	0.136676
Java Full Stack Developer	Deloitte	Honolulu	0.136205

Cluster 4

Job Title	Company	Location	Match Score
Business Information Mgmt Specialist (Initiative Data Mgmt/Consumer Banking Data) - Excel/SQL/Tableau/Python/Jira	TD Bank	Charlotte	0.103484
Software Developer – Java/PHP/Python	AIStrata Technology Solutions	Wilmington	0.102680
Sr. Java Front End developer	PETADATA	San Francisco	0.101860
Enterprise - Platform Engineer - Kubernetes, AWS, CI/CD	Erias Ventures, LLC	None	0.101547
Full stack Developer (Experience in Next JS & Nest JS)	The Dignify Solutions	Bentonville	0.100558

7. chatbot.py

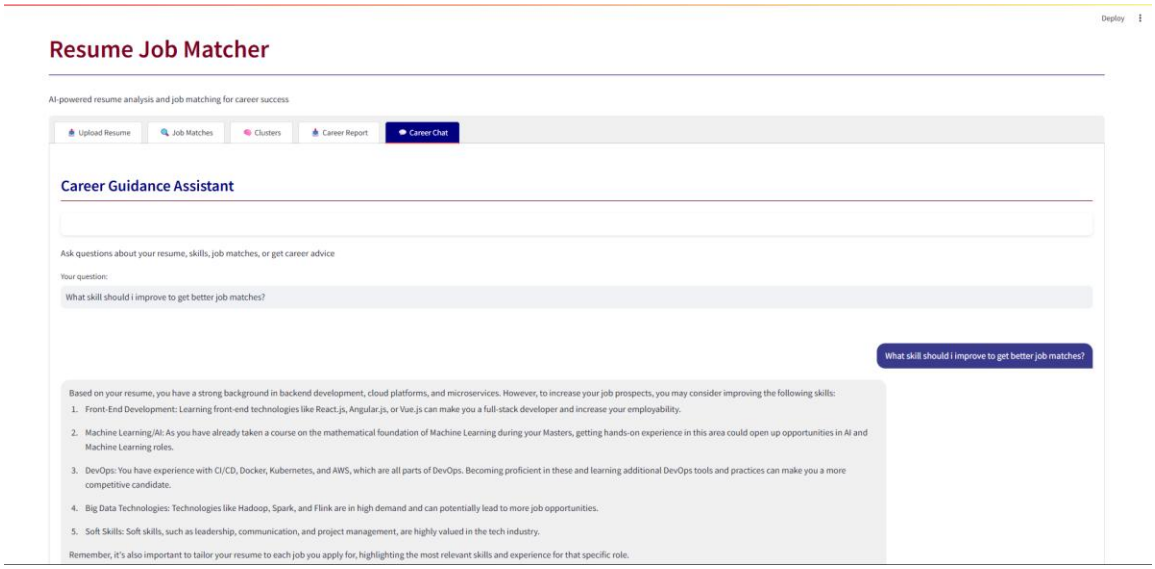
(Handles the chatbot logic. Supports both local (Flan-T5) and OpenAI GPT-3.5 based conversational responses.)

Code:

```
import os
import streamlit as st
import openai
import torch
from transformers import AutoTokenizer, AutoModelForSeq2SeqLM

# Load OpenAI key from Streamlit secrets
openai.api_key = st.secrets["openai"]["api_key"]

# Local model for fallback
_flant5_model = None
_tokenizer = None

def load_local_model():
    global _flant5_model, _tokenizer
    if _flant5_model is None or _tokenizer is None:
        _tokenizer = AutoTokenizer.from_pretrained("google/flan-t5-base")
```

```
        _flant5_model = AutoModelForSeq2SeqLM.from_pretrained("google/flan-t5-base")

def generate_chat_response(user_input, context=None):
    try:
        messages = [{"role": "system", "content": context}] if context else []
        messages.append({"role": "user", "content": user_input})

        response = openai.chat.completions.create(
            model="gpt-4",
            messages=messages,
            temperature=0.7,
            max_tokens=500
        )
        return response.choices[0].message.content

    except Exception as e:
        print("OpenAI API failed, using local model:", e)
        return generate_local_response(user_input, context)

def generate_local_response(prompt, context=None):
    load_local_model()
    final_prompt = f"Context: {context}\n\nQuestion: {prompt}" if context else prompt
    inputs = _tokenizer(final_prompt, return_tensors="pt", truncation=True)
    with torch.no_grad():
        outputs = _flant5_model.generate(**inputs, max_new_tokens=256)
    return _tokenizer.decode(outputs[0], skip_special_tokens=True)

def format_context(resume_text, matched_jobs=None):
    jobs_summary = "\n".join([f"- {j}" for j in matched_jobs]) if matched_jobs else ""
    return f"Resume:\n{resume_text}\n\nTop Matching Jobs:\n{jobs_summary}"
```

Output:



Lastly, this is the code of the final and the main script

Main_app.py

(The main Streamlit application script. It defines the UI, controls the tab layout (Upload, Match, Cluster, Report, Chat), handles state management, and orchestrates function calls from other components. All the above outputs were obtained by linking the function logic and outputs with the streamlit application)

Code:

```
import os
import warnings
import torch._dynamo

os.environ["TF_CPP_MIN_LOG_LEVEL"] = "3"
os.environ["TF_ENABLE_ONEDNN_OPTS"] = "0"
warnings.filterwarnings("ignore")
torch._dynamo.config.suppress_errors = True

import streamlit as st
from components.chatbot import generate_chat_response, format_context
from components.resume_parser import parse_resume_and_skills
from components.job_scraper import get_all_jobs
from components.job_matcher import rank_jobs_by_embedding
from components.job_clustering import cluster_jobs
from components.report_generator import generate_report

# --- Custom CSS ---
st.set_page_config(
    page_title="Resume Job Matcher",
    layout="wide",
    initial_sidebar_state="collapsed"
)

# Add custom CSS
st.markdown("""
<style>
    /* Main colors */
    :root {
        --burgundy: #800020;
        --navy: #000080;
        --light-burgundy: #aa5f73;
        --light-navy: #3a3a8c;
        --off-white: #f8f8f8;
        --light-gray: #f0f0f0;
    }

    /* Header styling */
    .main-header {
        color: var(--burgundy);
```

```
    font-weight: 600;
    padding-bottom: 20px;
    border-bottom: 2px solid var(--navy);
    margin-bottom: 30px;
}

/* Tabs styling */
.stTabs [data-baseweb="tab-list"] {
    gap: 10px;
    background-color: var(--light-gray);
    padding: 10px 10px 0 10px;
    border-radius: 8px 8px 0 0;
}

.stTabs [data-baseweb="tab"] {
    border-radius: 5px 5px 0 0;
    padding: 10px 20px;
    background-color: white;
    border: 1px solid #ddd;
    border-bottom: none;
}

.stTabs [data-baseweb="tab-panel"] {
    background-color: white;
    border: 1px solid #ddd;
    border-top: none;
    border-radius: 0 0 8px 8px;
    padding: 20px;
}

.stTabs [aria-selected="true"] {
    background-color: var(--navy) !important;
    color: white !important;
    font-weight: 600;
}

/* Button styling */
.stButton button {
    background-color: var(--burgundy);
    color: white;
    border-radius: 5px;
    padding: 5px 15px;
    border: none;
    font-weight: 600;
    transition: all 0.3s;
}

.stButton button:hover {
```

```
    background-color: var(--light-burgundy);
    box-shadow: 0 2px 5px rgba(0,0,0,0.2);
}

/* Card styling */
.card {
    background-color: white;
    border-radius: 8px;
    padding: 20px;
    box-shadow: 0 2px 5px rgba(0,0,0,0.1);
    margin-bottom: 20px;
}

/* Dataframe styling */
.dataframe th {
    background-color: var(--navy);
    color: white;
}

.dataframe tr:nth-child(even) {
    background-color: var(--light-gray);
}

/* Chat message styling */
.user-message {
    background-color: var(--light-navy);
    color: white;
    padding: 10px 15px;
    border-radius: 15px 15px 0 15px;
    margin: 10px 0;
    max-width: 80%;
    align-self: flex-end;
    display: inline-block;
}

.assistant-message {
    background-color: var(--light-gray);
    padding: 10px 15px;
    border-radius: 15px 15px 15px 0;
    margin: 10px 0;
    max-width: 80%;
    align-self: flex-start;
    display: inline-block;
}

/* Success message */
.success-banner {
    background-color: #d4edda;
```

```
    color: #155724;
    padding: 10px 15px;
    border-radius: 5px;
    border-left: 5px solid #28a745;
    margin: 10px 0;
}

/* Info/warning message */
.info-banner {
    background-color: #cce5ff;
    color: #004085;
    padding: 10px 15px;
    border-radius: 5px;
    border-left: 5px solid #0066ff;
    margin: 10px 0;
}

.warning-banner {
    background-color: #fff3cd;
    color: #856404;
    padding: 10px 15px;
    border-radius: 5px;
    border-left: 5px solid #ffc107;
    margin: 10px 0;
}

/* Badge styling */
.badge {
    display: inline-block;
    padding: 3px 10px;
    border-radius: 15px;
    font-size: 0.8em;
    margin-right: 5px;
    margin-bottom: 5px;
}

.badge-primary {
    background-color: var(--navy);
    color: white;
}

.badge-secondary {
    background-color: var(--burgundy);
    color: white;
}

/* Section styling */
.section-header {
```

```
        color: var(--navy);
        border-bottom: 2px solid var(--burgundy);
        padding-bottom: 10px;
        margin-top: 30px;
        margin-bottom: 20px;
    }
</style>
""", unsafe_allow_html=True)

# --- Streamlit Setup ---
st.markdown("<h1 class='main-header'>Resume Matcher Pro</h1>", unsafe_allow_html=True)
st.markdown("<p>AI-powered resume analysis and job matching for career success</p>",
unsafe_allow_html=True)

# --- Tabs ---
upload_tab, match_tab, clusters_tab, report_tab, chat_tab = st.tabs([
    "📤 Upload Resume", "🎯 Job Matches", "🧠 Clusters", "📥 Career Report", "💬 Career Chat"
])

# --- Global State ---
if 'resume_text' not in st.session_state:
    st.session_state.resume_text = ""
if 'skills' not in st.session_state:
    st.session_state.skills = []
if 'jobs_df' not in st.session_state:
    st.session_state.jobs_df = None
if 'ranked_df' not in st.session_state:
    st.session_state.ranked_df = None
if 'clustered_df' not in st.session_state:
    st.session_state.clustered_df = None

# --- Upload Tab ---
with upload_tab:
    st.markdown("<h3 class='section-header'>Upload Your Resume</h3>", unsafe_allow_html=True)

    col1, col2 = st.columns([3, 1])

    with col1:
        st.markdown("<div class='card'>", unsafe_allow_html=True)
        uploaded_file = st.file_uploader("Select your resume file (PDF or DOCX)", type=["pdf", "docx"])
        st.markdown("</div>", unsafe_allow_html=True)

    with col2:
        st.markdown("<div class='card'>", unsafe_allow_html=True)
        st.markdown("<p>Supported formats:</p>", unsafe_allow_html=True)
        st.markdown("• PDF<br>• DOCX", unsafe_allow_html=True)
        st.markdown("</div>", unsafe_allow_html=True)
```

```
    if uploaded_file:
        with st.spinner("Analyzing your resume..."):
            resume_text, extracted_skills = parse_resume_and_skills(uploaded_file)
            st.session_state.resume_text = resume_text
            st.session_state.skills = extracted_skills

        st.markdown("<div class='success-banner'>Resume successfully analyzed!</div>",
unsafe_allow_html=True)

        col1, col2 = st.columns([1, 1])

        with col1:
            st.markdown("<div class='card'>", unsafe_allow_html=True)
            st.markdown("<h4>Extracted Skills</h4>", unsafe_allow_html=True)
            skills_html = ""
            for skill in extracted_skills:
                skills_html += f"<span class='badge badge-primary'>{skill}</span>"
            st.markdown(skills_html, unsafe_allow_html=True)
            st.markdown("</div>", unsafe_allow_html=True)

        with col2:
            with st.expander(" 📄 Resume Text Preview"):
                st.text_area("", resume_text[:3000], height=200)

# --- Match Tab ---
with match_tab:
    st.markdown("<h3 class='section-header'>Find Your Perfect Job Match</h3>",
unsafe_allow_html=True)

    if st.session_state.resume_text and st.session_state.skills:
        st.markdown("<div class='card'>", unsafe_allow_html=True)
        if st.button(" 🔎 Search Jobs & Rank Matches"):
            with st.spinner("Searching for matching jobs..."):
                jobs_df = get_all_jobs(st.session_state.skills)
                ranked_df, missing = rank_jobs_by_embedding(
                    st.session_state.resume_text, jobs_df, st.session_state.skills
                )
                st.session_state.jobs_df = jobs_df
                st.session_state.ranked_df = ranked_df
                st.session_state.top_jobs = ranked_df.head(5).to_dict('records')

            st.markdown(f"<div class='success-banner'>Found {len(ranked_df)} potential job matches</div>",
unsafe_allow_html=True)

            if missing:
                missing_skills_html = ", ".join([f"<span class='badge badge-secondary'>{skill}</span>" for skill
in missing])
```

```
            st.markdown(f"<div class='warning-banner'>Top jobs require these skills that weren't found in your resume: {missing_skills_html}</div>", unsafe_allow_html=True)
        st.markdown("</div>", unsafe_allow_html=True)

        if st.session_state.ranked_df is not None:
            st.markdown("<h4>Top 10 Job Matches</h4>", unsafe_allow_html=True)

            # Safe column selection
            try:
                display_columns = []
                available_columns = st.session_state.ranked_df.columns.tolist()

                # Map expected columns to actual columns
                column_mapping = {
                    'Job Title': ['Job Title', 'job_title', 'title', 'position', 'role'],
                    'Company': ['Company', 'company', 'company_name', 'employer'],
                    'Location': ['Location', 'location', 'job_location', 'city'],
                    'Match Score': ['Match Score', 'match_score', 'score', 'similarity', 'match'],
                    'Apply Link': ['Apply Link', 'apply_link', 'url', 'link', 'job_url']
                }

                # Build columns to display
                display_df = st.session_state.ranked_df.copy()

                # Add columns if needed or rename existing ones
                for display_name, possible_names in column_mapping.items():
                    found = False
                    for col_name in possible_names:
                        if col_name in available_columns:
                            if col_name != display_name and col_name in display_df.columns:
                                display_df[display_name] = display_df[col_name]
                            display_columns.append(display_name)
                            found = True
                            break
                    if not found:
                        display_df[display_name] = "N/A"
                        display_columns.append(display_name)

                st.dataframe(
                    display_df[display_columns].head(10),
                    use_container_width=True,
                    hide_index=True
                )
            except Exception as e:
                st.error(f"Error displaying job matches: {e}")
                st.write("Available columns:", ", ".join(st.session_state.ranked_df.columns.tolist()))
    else:
```

```
        st.markdown("<div class='info-banner'>Please upload and analyze your resume first</div>",
unsafe_allow_html=True)

# --- Clusters Tab ---
# Inside the cluster analysis button logic in main_app.py (already structured):

with clusters_tab:
    st.markdown("<h3 class='section-header'>Job Clusters Analysis</h3>", unsafe_allow_html=True)

    if st.session_state.ranked_df is not None:
        st.markdown("<div class='card'>", unsafe_allow_html=True)
        st.markdown("<p>Group similar jobs together to identify career paths and opportunities</p>",
unsafe_allow_html=True)
        num_clusters = st.slider("Number of Job Clusters", 2, 6, 3)

        if st.button("🔍 Run Cluster Analysis"):
            try:
                with st.spinner("Running cluster analysis..."):
                    if 'Description' not in st.session_state.ranked_df.columns:
                        desc_alternatives = ['description', 'job_description', 'JobDescription']
                        found = False
                        for col in desc_alternatives:
                            if col in st.session_state.ranked_df.columns:
                                st.session_state.ranked_df['Description'] = st.session_state.ranked_df[col].fillna("")
                                found = True
                                break
                        if not found:
                            st.session_state.ranked_df['Description'] = st.session_state.ranked_df.apply(
                                lambda row: f"{row.get('Job Title', 'Unknown')} at {row.get('Company', 'Unknown')}",
axis=1
                            )
                    else:
                        st.session_state.ranked_df['Description'] =
st.session_state.ranked_df['Description'].fillna("")

                    clustered_df, fig = cluster_jobs(st.session_state.ranked_df, num_clusters=num_clusters)
                    st.session_state.clustered_df = clustered_df

                st.plotly_chart(fig, use_container_width=True)

                for cluster_id in sorted(clustered_df['Cluster'].unique()):
                    cluster_df = clustered_df[clustered_df['Cluster'] == cluster_id]

                    with st.expander(f"Cluster {cluster_id + 1} - {len(cluster_df)} jobs"):
                        try:
                            display_columns = []
                            available_columns = cluster_df.columns.tolist()
```

```
                    column_mapping = {
                        'Job Title': ['Job Title', 'job_title', 'title', 'position', 'role'],
                        'Company': ['Company', 'company', 'company_name', 'employer'],
                        'Location': ['Location', 'location', 'job_location', 'city'],
                        'Apply Link': ['Apply Link', 'apply_link', 'url', 'link', 'job_url']
                    }

                    display_df = cluster_df.copy()

                    for display_name, possible_names in column_mapping.items():
                        found = False
                        for col_name in possible_names:
                            if col_name in available_columns:
                                if col_name != display_name and col_name in display_df.columns:
                                    display_df[display_name] = display_df[col_name]
                                display_columns.append(display_name)
                                found = True
                                break
                        if not found:
                            display_df[display_name] = "N/A"
                            display_columns.append(display_name)

                    st.dataframe(
                        display_df[display_columns],
                        use_container_width=True,
                        hide_index=True
                    )
                except Exception as e:
                    st.error(f"Error displaying cluster data: {e}")
                    st.write("Available columns:", ", ".join(cluster_df.columns.tolist()))
            except Exception as e:
                st.error(f"Error during clustering: {e}")
        st.markdown("</div>", unsafe_allow_html=True)
    else:
        st.markdown("<div class='info-banner'>Please run the job match step first</div>",
unsafe_allow_html=True)

# --- Report Tab ---
with report_tab:
    st.markdown("<h3 class='section-header'>Career Opportunities Report</h3>",
unsafe_allow_html=True)

    if st.session_state.clustered_df is not None:
        st.markdown("<div class='card'>", unsafe_allow_html=True)
        st.markdown("""
        <p>Generate a comprehensive career report with:</p>
        <ul>
            <li>Resume analysis summary</li>
```

```
            <li>Top skill matches and gaps</li>
            <li>Personalized job recommendations</li>
            <li>Career path visualization</li>
        </ul>
        """, unsafe_allow_html=True)

        if st.button("📥 Generate Career Report"):
            try:
                with st.spinner("Creating your personalized career report..."):
                    report_path = generate_report(
                        st.session_state.resume_text,
                        st.session_state.skills,
                        st.session_state.ranked_df,
                        st.session_state.clustered_df
                    )

                st.markdown("<div class='success-banner'>Career report generated successfully!</div>",
unsafe_allow_html=True)

                with open(report_path, "rb") as f:
                    st.download_button(
                        "⬇️ Download Career Report",
                        f,
                        file_name="career_report.html",
                        use_container_width=True
                    )
            except Exception as e:
                st.error(f"Error generating report: {e}")
        st.markdown("</div>", unsafe_allow_html=True)
    else:
        st.markdown("<div class='info-banner'>Complete clustering first to unlock the report</div>",
unsafe_allow_html=True)

# --- Chat Tab ---
with chat_tab:
    st.markdown("<h3 class='section-header'>Career Guidance Assistant</h3>", unsafe_allow_html=True)

    st.markdown("<div class='card'>", unsafe_allow_html=True)
    st.markdown("<p>Ask questions about your resume, skills, job matches, or get career advice</p>",
unsafe_allow_html=True)

    if 'chat_history' not in st.session_state:
        st.session_state.chat_history = []

    # Chat input
    user_input = st.text_input("Your question:", placeholder="e.g., What skills should I improve to get better
job matches?")
```

```
    if user_input:
        try:
            with st.spinner("Thinking..."):
                context = format_context(st.session_state.resume_text,
matched_jobs=st.session_state.get("top_jobs", []))
                response = generate_chat_response(user_input, context)
                st.session_state.chat_history.append(("You", user_input))
                st.session_state.chat_history.append(("Assistant", response))
        except Exception as e:
            st.error(f"Error generating response: {e}")
            st.session_state.chat_history.append(("You", user_input))
            st.session_state.chat_history.append(("Assistant", "I'm sorry, I encountered an error while
processing your request."))
    st.markdown("</div>", unsafe_allow_html=True)

    # Display chat history
    st.markdown("<div style='margin-top: 20px;'>", unsafe_allow_html=True)
    for sender, message in st.session_state.chat_history:
        if sender == "You":
            st.markdown(f"<div style='text-align: right;'><div class='user-message'>{message}</div></div>",
unsafe_allow_html=True)
        else:
            st.markdown(f"<div><div class='assistant-message'>{message}</div></div>",
unsafe_allow_html=True)
    st.markdown("</div>", unsafe_allow_html=True)
```