Leveraging the power of OpenAI’s advanced language models, this project provides an automated and intelligent way to generate, refine, and match job descriptions with relevant job requirements. The system’s semantic matching capabilities ensure that job descriptions are tailored to specific roles, improving the accuracy and relevance of the information provided. Additionally, the integrated API creation feature allows seamless access to these functionalities, making it easier for businesses to incorporate the tool into their existing systems.

This solution is particularly beneficial for HR departments, recruitment agencies, and companies that need to manage large volumes of job descriptions. By automating the process, the AI-Powered Job Description Processor reduces the time and effort required to create high-quality job descriptions, while also enhancing the precision of candidate matching. The tool’s flexibility and scalability make it an invaluable asset for any organization looking to streamline its recruitment processes and ensure that job descriptions are always up-to-date and aligned with the latest industry standards.

A diagram of a software process

Description automatically generated

**Figure 1: The schematic concept of integrating OpenAI, FastAPI, a database, Docker, and Git for the job description generation process.**

**The project focuses on a FastAPI-based application that generates job descriptions and requirements using OpenAI’s language models. By converting job-related text into semantic vectors, the API produces relevant content for specified job titles. Containerization with Docker ensures consistent deployment.**

This dataset offers a comprehensive collection of job listings, specifically curated to bolster research and analysis in pivotal areas such as labor market trends, natural language processing (NLP), and machine learning. Conceived and compiled with a strong focus on educational and research endeavors, it showcases a broad spectrum of job advertisements across a diverse range of industries and job categories.

A screenshot of a computer

Description automatically generated

**Figure 2: The table for structuring the required data for embedding and job description generation.**

The command docker-compose up -d gitea mysql is used to start the Gitea and MySQL services in the background. This ensures that the necessary components for the project, such as version control (Gitea) and the database (MySQL), are up and running, ready to support the application.

(base) [KAdmin@Kubra mlops\_docker]$ docker-compose up -d gitea mysql



The steps under the Git commands involve creating a new directory (jobdescription), initializing it as a Git repository with git init, adding a README file, committing it with a message "this is test commit", and then pushing the changes to a remote repository hosted on Jenkins. This process sets up version control for the project, allowing for tracking changes and collaboration.

cd ..  
mkdir jobdescription  
cd jobdescription  
  
touch README.md  
git init  
git add README.md  
git commit -m "this is test commit"  
  
  
git remote add origin http://localhost:3000/jenkins/jobdescriptionApplication.git  
git push -u origin master   
  
# Username and password will be required

The photo shows that the initial commit, labeled “this is test commit,” has successfully been pushed to the Gitea repository for the jobdescriptionApplication project. This confirms that the project’s repository on Gitea is properly set up and accessible, allowing for version control and collaborative development within the Jenkins CI/CD pipeline.

A screenshot of a computer

Description automatically generated

The repository is being pulled from Gitea into PyCharm, allowing for seamless synchronization of the local project with the remote repository.

A screenshot of a computer

Description automatically generated

Connected to MySQL in the mlflow\_db Docker container and created a new database named position. A user was created and granted privileges on this database to manage its contents. Finally, the connection was tested to ensure everything was set up correctly.

# Connect mysql container   
docker exec -it mlflow\_db mysql -u root -p

A screenshot of a computer program

Description automatically generated

# Create database   
create database position;  
  
# Create user   
CREATE USER 'username'@'%' IDENTIFIED BY 'password';  
  
# Grand mlops\_user on mlops database   
GRANT ALL PRIVILEGES ON position.\* TO 'username'@'%' WITH GRANT OPTION;  
  
FLUSH PRIVILEGES;  
  
exit  
  
# Test connection   
  
docker exec -it mlflow\_db mysql -u username -D position -p  
  
exit

The MySQL command show databases; confirmed the successful creation of the position database along with other existing databases.

A screen shot of a computer

Description automatically generated

**Now let’s prepare the Python code for the project:**

The jobdescriptionApplication/ directory structure is organized to efficiently manage the components of the project. Below is a brief overview of the key files and folders:

* **main.py**: The entry point of the application, containing the main FastAPI app logic.
* **Database.py**: Handles database operations, including the creation of tables and database sessions.
* **function.py**: Contains utility functions, such as those for processing and cleaning job descriptions.
* **requirement.txt**: Lists the Python dependencies required to run the project.
* **DockerFile**: A Docker configuration file to containerize the application.
* **.env**: Stores environment variables such as database URLs and API keys.
* **.gitignore**: Specifies files and directories to be ignored by Git.
* **templates/**: Directory containing HTML templates for the web interface.
* **index.html**: The homepage template where users input job titles.
* **result.html**: Displays the processed job descriptions and requirements.
* **job\_description.csv**: The dataset file containing job descriptions to be processed.

A screen shot of a computer

Description automatically generated

**Requirements.txt**

openai  
pandas  
numpy  
scikit-learn  
uvicorn  
fastapi  
SQLAlchemy  
pymysql  
python-dotenv  
sqlmodel  
cryptography  
jinja2  
python-multipart

**.gitignore**

.env

**.env**

SQLALCHEMY\_DATABASE\_URL="mysql+pymysql://username:password@localhost/position"  
  
OPENAI\_API\_KEY=your-openai-api-key

**Dockerfile**

FROM python:3.9-slim  
COPY requirements.txt requirements.txt  
RUN pip install --upgrade pip  
RUN pip install -r requirements.txt  
COPY . /opt/  
WORKDIR /opt  
EXPOSE 8000  
ENTRYPOINT uvicorn main:app --host=0.0.0.0 --port=8000

**function.py**

import openai  
import pandas as pd  
import numpy as np  
from sklearn.metrics.pairwise import cosine\_similarity  
import re  
import os  
from dotenv import load\_dotenv  
  
# Load the .env file  
load\_dotenv()  
  
# Get the API key from the environment variable  
api\_key = os.getenv("OPENAI\_API\_KEY")  
  
# Initialize the OpenAI client  
client = openai.OpenAI(api\_key=api\_key)  
  
  
def get\_embedding(text, model="text-embedding-3-small"):  
 text = text.replace("\n", " ")  
 return client.embeddings.create(input=[text], model=model).data[0].embedding  
  
def clean\_paragraph(paragraph):  
 if not isinstance(paragraph, str):  
 return "" # Return an empty string if the input is not a string  
  
 patterns\_to\_remove = [  
 r'Resource Innovations \(RI\)',  
 r'a women-led',  
 r'focused on impact',  
 r'currently reside in the San Francisco CA Bay area',  
 r'visa sponsorship or extensions',  
 r'not offering',  
 r'unfortunately',  
 r'We require candidates to',  
 r'With every step, we’re leading the charge to power change\.',  
 r'Building on our expertise in',  
 r'Description Ryanair Labs is the technology brand of Ryanair.',  
 r'Labs is a state of-the-art digital & IT innovation',  
 r'^\s\*Description\s\*',  
 r'Methods is a.\*?\.',  
 r'About the role:',  
 r'What Methods offer is to you:.\*',  
 r'\bRyanair\b',  
 r'responsible for\b.\*',  
 r'Resource Innovations \(RI\)',  
 r'a women-led',  
 r'focused on impact',  
 r'currently reside in the San Francisco CA Bay area',  
 r'visa sponsorship or extensions',  
 r'not offering',  
 r'unfortunately',  
 r'We require candidates to',  
 r'With every step, we’re leading the charge to power change\.',  
 r'Building on our expertise in',  
 r'Description Ryanair Labs is the technology brand of Ryanair.',  
 r'Labs is a state of-the-art digital & IT innovation',  
 r'The Recruitment Process',  
 r'Xplore has retained Bert Sadtler, President of Boxwood Strategies to manage all hiring',  
 r'While this process is different, we feel that it genuinely offers both candidates and the best opportunity',  
 r'Please click for a short video on the Boxwood Process',  
 r'COMPENSATION.\*?(?=\s[A-Z])',  
 r'The Recruitment Process.\*?(?=\s[A-Z])',  
 r'While this process is different.\*?(?=\s[A-Z])',  
 r'Boxwood Process.\*?(?=\s[A-Z])'  
 r'SmartLogic '  
  
 ]  
  
 cleaned\_text = paragraph  
 for pattern in patterns\_to\_remove:  
 cleaned\_text = re.sub(pattern, '', cleaned\_text)  
  
 cleaned\_text = re.sub(r'\s+', ' ', cleaned\_text).strip()  
  
 return cleaned\_text  
  
def combine\_requirements(df):  
 # Combine 'Requirement' and 'Requirements' columns into 'Combined\_Requirements'  
 df['Combined\_Requirements'] = (df['Requirement'].fillna('') + ' ' + df['Requirements'].fillna('')).str.strip()  
  
 # Drop the original columns  
 df.drop(columns=['Requirement', 'Requirements'], inplace=True)  
  
 return df  
  
def find\_most\_similar\_job\_details(df, new\_job\_titles):  
 job\_titles = df['Category'].tolist()  
 descriptions = df['Description'].tolist()  
 requirements = df['Combined\_Requirements'].tolist()  
 similar\_details = {}  
  
 for new\_job\_title in new\_job\_titles:  
 embeddings = [get\_embedding(title) for title in job\_titles]  
 embeddings.append(get\_embedding(new\_job\_title))  
  
 # Calculate cosine similarity  
 numpy\_array = np.array(embeddings)  
 new\_job\_title\_embedding = numpy\_array[-1].reshape(1, -1)  
 job\_title\_embeddings = numpy\_array[:-1]  
  
 similarities = cosine\_similarity(new\_job\_title\_embedding, job\_title\_embeddings)[0]  
 most\_similar\_index = np.argmax(similarities)  
  
 # Clean up the description and requirements  
 cleaned\_description = clean\_paragraph(descriptions[most\_similar\_index])  
 cleaned\_requirements = clean\_paragraph(requirements[most\_similar\_index])  
  
 # Shorten the description if necessary (e.g., only the first 2 sentences)  
 cleaned\_description = '. '.join(cleaned\_description.split('. ')[:2])  
  
 similar\_details[new\_job\_title] = {  
 'description': cleaned\_description,  
 'requirements': cleaned\_requirements  
 }  
  
 return similar\_details

**models.py**

from sqlmodel import SQLModel, Field  
from typing import Optional  
  
class JobDetails(SQLModel, table=True):  
 id: Optional[int] = Field(default=None, primary\_key=True)  
 category: str  
 description: str

**Database.py**

import os  
from sqlmodel import SQLModel, Field, create\_engine, Session  
from typing import Optional  
import pymysql  
from dotenv import load\_dotenv  
from sqlalchemy import LargeBinary  
  
# Load the .env file  
load\_dotenv()  
  
# Get the database URL from the .env file  
DATABASE\_URL = os.getenv("SQLALCHEMY\_DATABASE\_URL")  
  
# Create the database engine  
engine = create\_engine(DATABASE\_URL, echo=True)  
  
# JobDetails table definition with LONGTEXT for large text fields  
class JobDetails(SQLModel, table=True):  
 id: Optional[int] = Field(default=None, primary\_key=True)  
 category: str  
 description: str = Field(sa\_column=LargeBinary()) # Use LONGTEXT to store very large descriptions  
 requirements: str = Field(sa\_column=LargeBinary()) # Use LONGTEXT to store very large requirements  
  
# Function to create the database and tables  
def create\_db\_and\_tables():  
 SQLModel.metadata.create\_all(engine)  
  
# Function to get the database session  
def get\_session():  
 with Session(engine) as session:  
 yield session  
  
# Function to insert job details into the database  
def insert\_job\_details(session: Session, category: str, description: str, requirements: str):  
 try:  
 job\_details = JobDetails(category=category, description=description, requirements=requirements)  
 session.add(job\_details)  
 session.commit()  
 session.refresh(job\_details)  
 return job\_details  
 except Exception as e:  
 print(f"Error inserting into database: {e}")  
 session.rollback()  
 raise

**index.html**

<!DOCTYPE html>  
<html lang="en">  
<head>  
 <meta charset="UTF-8">  
 <meta name="viewport" content="width=device-width, initial-scale=1.0">  
 <title>Job Description Generator</title>  
</head>  
<body>  
 <h1>Job Description Generator</h1>  
 <form action="/process-csv/" method="post">  
 <label for="category">Select Job Category:</label>  
 <select id="category" name="category">  
 {% for category in categories %}  
 <option value="{{ category }}">{{ category }}</option>  
 {% endfor %}  
 </select>  
 <br><br>  
 <input type="submit" value="Generate Description">  
 </form>  
</body>  
</html>

**result.html**

<!DOCTYPE html>  
<html lang="en">  
<head>  
 <meta charset="UTF-8">  
 <meta name="viewport" content="width=device-width, initial-scale=1.0">  
 <title>Generated Job Details</title>  
</head>  
<body>  
 <h1>Generated Job Details</h1>  
 <ul>  
 {% for i in range(descriptions | length) %}  
 <li><strong>Description:</strong> {{ descriptions[i] }}</li>  
 <li><strong>Requirements:</strong> {{ requirements[i] }}</li>  
 {% endfor %}  
 </ul>  
</body>  
</html>

**main.py**

from fastapi import FastAPI, Request, Form, Depends  
from fastapi.responses import HTMLResponse  
from fastapi.templating import Jinja2Templates  
import pandas as pd  
from function import find\_most\_similar\_job\_details, combine\_requirements  
from Database import create\_db\_and\_tables, get\_session, insert\_job\_details  
from sqlmodel import Session  
  
app = FastAPI()  
  
templates = Jinja2Templates(directory="templates")  
  
@app.on\_event("startup")  
def on\_startup():  
 create\_db\_and\_tables()  
  
@app.get("/", response\_class=HTMLResponse)  
async def read\_form(request: Request):  
 df = pd.read\_csv("job\_description.csv", index\_col=0)  
 categories = df['Category'].unique().tolist()  
 return templates.TemplateResponse("index.html", {"request": request, "categories": categories})  
  
@app.post("/process-csv/", response\_class=HTMLResponse)  
async def process\_csv(request: Request, category: str = Form(...), session: Session = Depends(get\_session)):  
 df = pd.read\_csv("job\_description.csv", index\_col=0)  
 df = combine\_requirements(df) # Combine 'Requirement' and 'Requirements'  
  
 similar\_details = find\_most\_similar\_job\_details(df, [category])  
  
 for details in similar\_details.values():  
 insert\_job\_details(session, category, details['description'], details['requirements'])  
  
 # Pass the zip function to the template context  
 return templates.TemplateResponse("result.html", {  
 "request": request,  
 "descriptions": [details['description'] for details in similar\_details.values()],  
 "requirements": [details['requirements'] for details in similar\_details.values()],  
 "zip": zip  
 })

This sequence of commands demonstrates how to set up and run the jobdescriptionApplication. First, a new Python environment (Python 3.9) is created and activated. The project repository is then cloned from a Gitea server. After moving into the project directory, dependencies are installed using the requirements.txt file. Finally, the FastAPI server is launched using Uvicorn, making the application accessible on the specified port (8002).

conda create --name myenv python=3.9

conda activate myenv  
git clone http://localhost:3000/jenkins/jobdescriptionApplication.git  
# You should go to jobdescriptionApplication file  
pip install -r requirements.txt  
  
uvicorn main:app --host 0.0.0.0 --port 8002 --reload

After running these commands, you can start testing the FastAPI application by accessing the documentation and interactive API testing interface at <http://localhost:8002/docs>. Navigate to the interactive documentation, and you can easily test the /process-csv/ endpoint. Simply enter the desired job category (e.g., 'software developer') in the input field and click 'Execute' to trigger the API and generate the corresponding job description content.

A screenshot of a computer

Description automatically generated

This image demonstrates a successful API request made to the /process-csv/ endpoint using FastAPI. The response body shows the generated job details, including a description and requirements for the specified job category. The request URL, response code, and response headers are also displayed, indicating a correct execution of the API.

A screenshot of a computer

Description automatically generated

***Now, let’s check the database to verify if the data has been correctly inserted.***

docker exec -it mlflow\_db mysql -u username -D position -p  
USE position;  
SELECT \* FROM JobDetails;

A black background with white text

Description automatically generated

**Conclusion**

This project demonstrates the integration of FastAPI with advanced AI-powered tools, leveraging OpenAI for semantic text processing and Docker for streamlined deployment. By developing a custom API that processes job descriptions based on specific categories, the system effectively generates tailored job descriptions and requirements. The implementation highlights the use of modern development practices, including containerization and API development, to create a flexible and efficient tool for automating and enhancing job description management. This approach not only simplifies the process but also ensures accuracy and relevance in job postings, making it a valuable asset for organizations aiming to optimize their recruitment processes.

***Thanks for reading!***