Here’s a step-by-step guide to building your **TDS Solver** project:

**Step 1: Understand the Project Requirements**

* **Goal:** Create an API that takes a **graded assignment question** and responds with an **answer**.
* **Input:** A question (string) and optional file attachments (e.g., ZIP files).
* **Output:** A JSON object with the correct answer.
* **Deployment:** API must be publicly accessible (e.g., Vercel, Render, Railway, etc.).
* **Evaluation:** The API will be tested with five random questions.

**Step 2: Plan the Tech Stack**

**Core Components**

* **Backend Framework:** FastAPI (Python) for handling API requests.
* **LLM Integration:** OpenAI’s GPT model or a custom fine-tuned model.
* **File Processing:** Extract data from attached files (ZIP, CSV, TXT, etc.).
* **Deployment:** Use Vercel, Render, or Railway.
* **Database (Optional):** Store previous questions/answers for caching.

**Tools Needed**

* Python (with FastAPI)
* OpenAI API (or another LLM API)
* Docker (optional for containerization)
* GitHub for version control
* Postman or curl for testing

**Step 3: Set Up the FastAPI Application**

**1. Create a new project folder**

bash

CopyEdit

mkdir tds-solver && cd tds-solver

**2. Set up a virtual environment**

bash

CopyEdit

python3 -m venv env

source env/bin/activate # For Windows, use: env\Scripts\activate

**3. Install dependencies**

bash

CopyEdit

pip install fastapi uvicorn python-multipart openai

**4. Create app.py**

python

CopyEdit

from fastapi import FastAPI, File, UploadFile, Form

import openai

import os

import zipfile

import pandas as pd

app = FastAPI()

# Set your OpenAI API key

openai.api\_key = os.getenv("OPENAI\_API\_KEY")

@app.post("/api/")

async def solve\_assignment(question: str = Form(...), file: UploadFile = None):

# Handle file if provided

if file:

file\_path = f"uploads/{file.filename}"

with open(file\_path, "wb") as f:

f.write(file.file.read())

# If the file is a ZIP, extract it

if file.filename.endswith(".zip"):

with zipfile.ZipFile(file\_path, 'r') as zip\_ref:

zip\_ref.extractall("extracted\_files")

extracted\_csv = [f for f in os.listdir("extracted\_files") if f.endswith(".csv")]

if extracted\_csv:

df = pd.read\_csv(f"extracted\_files/{extracted\_csv[0]}")

if "answer" in df.columns:

return {"answer": str(df["answer"].iloc[0])}

# Process question with OpenAI API

response = openai.ChatCompletion.create(

model="gpt-4o-mini",

messages=[{"role": "user", "content": question}]

)

return {"answer": response["choices"][0]["message"]["content"]}

**Step 4: Test the API Locally**

Run the server:

bash

CopyEdit

uvicorn app:app --reload

Test it with curl:

bash

CopyEdit

curl -X POST "http://127.0.0.1:8000/api/" \

-F "question=What is 2+2?"

**Step 5: Deploy the API**

**Option 1: Deploy on Vercel**

1. Install Vercel CLI:

bash

CopyEdit

npm install -g vercel

1. Login to Vercel:

bash

CopyEdit

vercel login

1. Deploy the project:

bash

CopyEdit

vercel

**Option 2: Deploy on Render**

1. Create a **new Render Web Service**.
2. Connect your GitHub repo.
3. Use the following **start command**:

bash

CopyEdit

uvicorn app:app --host 0.0.0.0 --port $PORT

1. Deploy!

**Step 6: Publish the GitHub Repository**

1. **Initialize Git and create a repo**

bash

CopyEdit

git init

git add .

git commit -m "Initial commit"

git branch -M main

git remote add origin https://github.com/YOUR\_USERNAME/tds-solver.git

git push -u origin main

1. **Add an MIT License**

bash

CopyEdit

echo "MIT License" > LICENSE

git add LICENSE

git commit -m "Added MIT License"

git push

**Step 7: Submit the API & GitHub URL**

* **Submit your GitHub repo and API endpoint URL** [in the Google Form](https://forms.gle/6ZLCGEEHUHVK71Yu5).
* Make sure your API **stays live** until results are announced.

🚀 **Follow these steps, and you'll have a fully functional, deployed TDS Solver API!** Let me know if you need further guidance. 🚀

Top of Form



O

Search

Reason

**Another approach : dt. 7-Mar-2025**

I want to develop an app that answers the questions asked in previous graded assignments (GAs) for the subject “Tools in Data Science”. There are total approx.60 different types of questions as listed in attached file. Project in detail is as below .

Background You are a clever student who has joined IIT Madras’ Online Degree in Data Science. You have just enrolled in the Tools in Data Science course.

To make your life easier, you have decided to build an LLM-based application that can automatically answer any of the graded assignment questions.

Specifically, you are building and deploying an API that accepts any question from one of these 5 graded assignments:

Graded Assignment 1 Graded Assignment 2 Graded Assignment 3 Graded Assignment 4 Graded Assignment 5 … and responds with the answer to enter in the assignment.  
Create an API

Your application exposes an API endpoint. Let’s assume that it is at <https://your-app.vercel.app/api/>.

The endpoint must accept a POST request, e.g. POST https://your-app.vercel.app/api/ with the question as well as optional file attachments as multipart/form-data.

For example, here’s how anyone can make a request:

curl –X POST “https://your-app.vercel.app/api/” \

-H “Content-Type: multipart/form-data” \

-F “question=Download and unzip file abcd.zip which has a single extract.csv file inside. What is the value in the “answer” column of the CSV file?” \

-F “file=@abcd.zip”

Copy to clipboardErrorCopied

The response must be a JSON object with a single text (string) field: answer that can be directly entered in the assignment. For example:

{

“answer”: “1234567890”

}

Copy to clipboardErrorCopied

Deploy your application

Deploy your application to a public URL that can be accessed by anyone. You may use any platform, including Vercel.

(If you use ngrok, ensure that it is running continuously until you get your results.)

Share your code

Create a new public GitHub repository

Add an MIT LICENSE file

Commit and push your code

Submit your URL

Submit your GitHub repository URL and your API endpoint URL in this Google Form: <https://forms.gle/6ZLCGEEHUHVK71Yu5>.

Evaluation

Pre-requisites: Your repository MUST meet the following criteria to be eligible for evaluation

Your GitHub repository exists and is publicly accessible

Your GitHub repository has a LICENSE file with the MIT license

We will send 5 questions randomly chosen from the graded assignments above. Correct answers will be awarded 4 marks each.

**ChatGPT response :**

Here are different approaches you can take to build your LLM-based assignment answering app:

**1. Fine-Tuning an LLM with Graded Assignment Data**

**Approach:**

* Collect all previous graded assignment questions and their answers.
* Fine-tune an open-source model like **Llama 3**, **Mistral**, or **GPT-J** with this dataset.
* Deploy the fine-tuned model as an API.

**Pros:** ✔️ High accuracy for known questions.  
✔️ Fast inference once fine-tuned.  
✔️ Can work offline after deployment.

**Cons:** ❌ Requires GPU resources for training.  
❌ Cannot generalize well to unseen questions.

**Implementation Stack:**

* **Model:** Llama 3 / GPT-3.5 Turbo / Mistral
* **Fine-tuning:** Hugging Face Transformers + PEFT (Parameter-Efficient Fine-Tuning)
* **Deployment:** FastAPI + Docker + Vercel

**2. Using GPT-4o or Claude with RAG (Retrieval-Augmented Generation)**

**Approach:**

* Store past graded assignment questions & answers in a **vector database** (e.g., **ChromaDB**, **FAISS**, **Weaviate**).
* When a new question is asked, retrieve similar questions using embeddings.
* Provide the retrieved questions as context to **GPT-4o** or **Claude** for generating an answer.

**Pros:** ✔️ Handles unseen questions by leveraging contextual retrieval.  
✔️ No need to fine-tune a model.  
✔️ Can integrate file processing for CSV/ZIP-based questions.

**Cons:** ❌ Slightly slower due to retrieval step.  
❌ Requires a good chunk of storage for embeddings.

**Implementation Stack:**

* **Vector Database:** FAISS / ChromaDB / Pinecone
* **Embedding Model:** OpenAI's text-embedding-3-small / BAAI-BGE
* **LLM API:** GPT-4o / Claude / Gemini
* **Backend:** FastAPI / Flask
* **Deployment:** Vercel / AWS Lambda

**3. Prompt Engineering with GPT-4o and Tool Use**

**Approach:**

* Design prompts that dynamically adjust based on the question type.
* If a file is attached (e.g., ZIP, CSV), use Python tools like **pandas**, **zipfile**, and **httpx** to extract data before sending it to the model.
* GPT-4o will be instructed to:
  + Answer directly for text-based questions.
  + Provide code for coding-related questions.
  + Extract and process files for CSV-based questions.

**Pros:** ✔️ No need for model training.  
✔️ Highly flexible—handles both text and file-based inputs.  
✔️ Works with external tools like Bash, SQL, and Excel.

**Cons:** ❌ Depends on GPT-4o API availability.  
❌ More expensive per API call compared to fine-tuning.

**Implementation Stack:**

* **LLM API:** GPT-4o
* **File Handling:** pandas, zipfile, SQLAlchemy (for SQL questions)
* **Backend:** FastAPI / Flask
* **Deployment:** Vercel / AWS Lambda

**4. Hybrid Approach: RAG + Function Calling**

**Approach:**

* Use **GPT-4o Function Calling** to decide the best tool for answering the question.
* If a past answer exists, retrieve it using **RAG**.
* If code execution is needed (e.g., counting Wednesdays in a date range), the API calls **Python scripts** to compute the answer.
* If a file is uploaded, automatically process it before passing it to the LLM.

**Pros:** ✔️ Can handle **both factual recall** and **new problems** dynamically.  
✔️ More cost-efficient since it avoids LLM calls when possible.  
✔️ Easily scalable with additional tools.

**Cons:** ❌ Slightly more complex to implement.  
❌ Requires managing multiple external APIs.

**Implementation Stack:**

* **LLM API:** GPT-4o
* **Vector Database:** FAISS / ChromaDB
* **Function Calling:** OpenAI API for tool selection
* **Execution Engine:** Python (for data extraction & calculations)
* **Backend:** FastAPI
* **Deployment:** Vercel

**Which Approach to Choose?**

| **Approach** | **Best for** | **Cost** | **Complexity** |
| --- | --- | --- | --- |
| Fine-Tuning LLM | Frequent questions | Medium (training required) | High |
| RAG with GPT-4o | New & old questions | Medium (LLM API usage) | Medium |
| Prompt Engineering | Quick setup | High (LLM API calls) | Low |
| Hybrid (RAG + Function Calling) | Complex use cases | Medium | High |

If you want **quick deployment**, go for **Prompt Engineering + GPT-4o**.  
If you want **cost efficiency**, use **RAG + GPT-4o**.  
If you want **high accuracy for known questions**, fine-tune a model.

Do you want implementation details for a specific approach? 🚀

I want to go with below approach.

Fine-Tuning LLM.

I would like to use following credentials for some of the questions as per need :\

&#x20;

api\\_key = os.getenv("OPENAI\\_API\\_KEY") or os.getenv("AIPROXY\\_TOKEN")

if not api\\_key or api\\_key.strip() == "":

&#x20; raise ValueError("API key is not set!")

print("AIPROXY\\_TOKEN:", api\\_key[:5] + "..." + api\\_key[-5:]) # Partial print for security

url for chat completions&#x20;

url = '[https://aiproxy.sanand.workers.dev/openai/v1/chat/completions](https://aiproxy.sanand.workers.dev/openai/v1/chat/completions)'

url for embeddings :&#x20;

CHAT\\_PROXY\\_URL = [https://aiproxy.sanand.workers.dev/openai/v1/embeddings](https://aiproxy.sanand.workers.dev/openai/v1/embeddings)

Model of openai to be used for embeddings api :&#x20;

text-embedding-3-small\

\

I will be using vs code with wsl to build this app. Give me the code for this now Bottom of Form