Assignment: Discourse Q&A Assistant with RAG (Using OpenRouter via AIPipe)

1. Problem Statement

Your task is to build a **Question–Answering web application** that helps students and instructors get answers from a **Discourse forum** using **Retrieval-Augmented Generation (RAG)**.

The system must:

1. **Fetch and index data** from a Discourse forum (topics, posts, replies).
2. Use a **RAG pipeline** where:
    - A retriever fetches relevant posts from your index.
    - A generator (LLM) answers the user's query using those retrieved posts as context.
3. Use **OpenAI-compatible models via OpenRouter, through https://aipipe.org/**.
4. Expose a **backend API** supporting RAG.
5. Provide a **clean, user-friendly frontend** for interacting with the assistant.
6. Include a **short demo video** of your application.

You may work in teams of **2–4** students (adjust as per your course rules).

---

2. Learning Objectives

By the end of this assignment, you should be able to:

- Use the **Discourse API** (or exported JSON) to collect forum data programmatically.
- Clean, structure, and **index text data** for retrieval.
- Design and implement a **RAG pipeline** (retriever + LLM).

- Integrate **OpenRouter LLMs via AIPipe** in a backend service.
- Build a **full-stack web application** (frontend + backend) with good UX.
- Communicate your design and implementation clearly in a **video demo**.

---

3. Functional Requirements

3.1 Data Fetching from Discourse

Your application must:

1. Fetch **at least N topics** and their **posts/replies** from a Discourse forum.
    - N = 500 posts minimum from `reading-club`.
2. Store the fetched data in a suitable format (e.g. database + vector store).

Minimum expectations:

- Fetch title, content/body, author username, created_at timestamp, and topic URL.
- Deduplicate content and handle long posts (chunk them if needed).

3.2 RAG Backend

You must build a backend (FastAPI / Flask / Node / Django / etc.) that exposes APIs for:

1. `POST /ask`
    - Request: `{ "question": "...", "filters": { ...optional... } }`

Response:

```
{
 "answer": "…generated answer…",
 "contexts": [
```

```
    {
      "post_id": "...",
      "topic_title": "...",
      "excerpt": "...",
      "url": "...",
      "score": 0.87
    }
  ]
}
```

        -

2. `GET /health`
    - Simple status endpoint to check if the service is running.
3. Optional:
    - `POST /reindex` to rebuild the index.
    - `GET /search` for pure retrieval without generation.

Backend must:

- Implement **RAG**:
    - Embed/encode chunks of Discourse posts into a vector database or in-memory index.
    - On each question, retrieve top-k relevant chunks.
    - Call an **OpenAI-compatible LLM via OpenRouter routed through AIPipe** with:
        - the question
        - retrieved chunks as context
- Ensure prompts **cite sources** and avoid hallucination when answer is not in the forum:
    - e.g., “I couldn’t find an answer in the forum posts. You may want to ask this as a new question.”

3.3 LLM Integration (OpenRouter via AIPipe)

- You must **not** call OpenAI directly.
- Instead, you must:
    - Configure a pipeline/project in **aipipe.org** that uses **OpenRouter models** (e.g., an OpenAI-compatible endpoint).
    - Use that AIPipe endpoint in your backend as if it were an OpenAI-style API:
        - Provide model name (if needed), prompt, and context.
- Keep your API keys **secret** (env variables or config files, not hard-coded in the repo).

You may choose any OpenRouter-supported LLM (e.g., GPT-4 class, high-quality smaller models, etc.), but you should mention your choice in the **README** and the **video**.

3.4 Frontend Requirements

Your frontend should be a **single-page web app** (React / Vue / Svelte / plain HTML+JS, etc.) with:

1. **Chat-like interface**:
    - Input box for the question.
    - Message area showing conversation history.
2. Display of:
    - Model's **answer**.
    - **Cited sources** (e.g., "Source 1, Source 2" with clickable links to the original Discourse posts).
3. Clear UI states:
    - Loading indicator while answer is being generated.
    - Error messages when backend/API fails.

4. Optional but strongly encouraged:
    - Filters (e.g., tags, time range, category).
    - Toggle between “Short answer” and “Detailed answer”.

Frontend should be **visually clean, responsive, and easy to use**. It doesn’t have to be fancy, but it should not look like a bare-bones debug UI.

---

4. Non-Functional Requirements

- **Code quality**:
    - Clear structure (modules, components, routes).
    - Comments where necessary.
    - A `README.md` explaining:
        - how to run backend
        - how to run frontend
        - how to set environment variables (API keys)
- **Security & privacy**:
    - No API keys committed to Git.
    - Use `.env` or config files ignored by version control.
- **Reproducibility**:
    - Provide sample `.env.example`.
    - Document any assumptions about the Discourse forum URL / API.

---

5. Deliverables

1. **Source Code Repository**
    - Backend code.
    - Frontend code.
    - Any scripts for data ingestion and indexing.
    - `README.md` with:
        - Project overview.
        - Architecture diagram (even a simple text or image is fine).
        - Setup instructions.
        - Example API calls.
2. **Running Application**
    - Either:
        - Deployed version (preferred: e.g., Render, Railway, Vercel, etc.), **or**
        - Clear instructions for running locally via `docker-compose` or simple commands.
3. **Demo Video (5–8 minutes)**
   The video should:
    - Introduce the problem:
        - “We built a Discourse Q&A assistant using RAG.”
    - Show the **UI in action**:
        - Ask 2–3 meaningful questions.
        - Show citations and how clicking them opens the corresponding forum posts.
    - Briefly explain the **architecture**:

        - Data flow: Discourse → ingestion script → vector store → backend → AIPipe (OpenRouter) → frontend.
    - Show a quick look at:
        - Where you call the LLM (code).
        - How you do retrieval (code or DB).
    - Mention:
        - Which model(s) you used via OpenRouter.
        - Known limitations and future improvements.
4. Format: upload to a platform (e.g., unlisted YouTube/GDrive) and share the link in your submission.

---

6. Suggested Tech Stack (Not Mandatory)

- **Backend**: FastAPI / Flask / Django / Node (Express/Nest)
- **Frontend**: React / Vue / Svelte
- **Vector Store**:
    - Simple: in-memory FAISS or similar.
    - Advanced: Qdrant, Chroma, Weaviate, or a cloud vector store.
- **LLM Access**:
    - OpenRouter models via AIPipe (OpenAI-compatible endpoint).