Application Design Document

1. Project Overview

Title: Global AI Colab For Good

Objective: The scope of this project is to build a global platform that links AI research groups with organizations aiming to solve social issues using AI. The platform will have a search interface for organizations to look for AI research papers relevant to their social cause. A dashboard will provide a curated list of relevant research to the user prompt, the research groups, and how the research work relates to the user's problem prompt. The platform will be designed to support a growing number of research groups and global organizations. We process a large corpus of AI research papers & social issue descriptions and train LLMs for information retrieval and matching between research and real-world problems.

2. Solution Architecture

High-Level Overview

The application consists of the following main components:

1. **Frontend**: A user-facing interface allowing organizations to input prompts and view research paper recommendations and explanations. Developed using Streamlit.
2. **Backend**: FastAPI-based API to manage interactions between the frontend and the processing modules.
3. **Data Processing Modules**:
   - **Data Scraping**: Retrieves AI research papers from ArXiv.
   - **Data Embedding & Storage**: Embeds paper content for search compatibility and stores it in ChromaDB.
   - **Retrieval-Augmented Generation (RAG) Module**: Processes user queries to retrieve and present relevant research paper insights.
4. **Database**:
   - **Vector Database (ChromaDB)**: Stores paper embeddings, enabling vector search.
5. **LLM Model Integration**: Uses finetuned LLM to match user queries to relevant papers and generate interpretative responses.
6. **Container Orchestration**: Docker-Compose orchestrates containers for seamless workflow.

Component Interactions

- The **frontend** interacts with the **backend API** for data submission and retrieval.

- The **backend API** connects to the **RAG module** for processing queries and retrieving responses.
- The **RAG module** interfaces with the **vector database** to retrieve relevant research embeddings and uses LLMs for response generation.
- **Data scraping and embedding modules** work independently to update the vector database, ensuring the most relevant content for user queries.

3. Technical Architecture

Technologies and Frameworks

- **Frontend**: We use Streamlit for the user interface.
- **Backend**: FastAPI for handling HTTP requests and integrating APIs.
- **Containers**: Docker for containerization of individual components, ensuring modularity and scalability.
- **Orchestration**: We have an automated build of Docker containers for each module using one Docker-Compose script.
- **Database**: ChromaDB for vector storage of embedded research papers.
- **LLM Integration**: We have two points of LLM integration. 1. We fine-tune a relevance rating LLM (with the base as Gemini-1.5-Flash API) which rates papers as relevant/not-relevant and is used as a filtration step after retrieving papers similar to the user's query. 2. We use a base model (Gemini-1.5-Flash API) which is fed the filtered papers in-context and prompted to provide a useful response for the non-profit user. We use VertexAI for fine-tuning and for content generation.
- **Vector Database:** We use ChromaDB with LangChain as our choice of vector database for the research papers. The database is stored on the cloud.
- **Data Embedding**: Hugging Face's Sentence Transformers (`all-MiniLM-L6-v2`) for embedding paper text.

Design Patterns

- **Microservices Architecture**: Each function (scraping, embedding, RAG) is modular and can scale independently.
- **API Gateway Pattern**: FastAPI acts as an interface layer between the frontend and backend components, ensuring smooth data flow.
- **Containerization Pattern**: Each processing task is containerized for flexibility and reproducibility.

4. APIs & Frontend Implementation

API Design (FastAPI)

`GET /`:

- **Description**: Root endpoint that serves as a welcome message.

- **Response**: Returns a JSON message `{ "message": "Welcome to AC215" }`.

`POST /api/perform_rag`:

- **Description**: Accepts a user query, performs the full Retrieve and Generate (RAG) process, and returns a summarized answer with relevant research paper snippets.

Request Body:

```
{

  "query": "string"  // The user's search query

}
```

Response: Returns a JSON object containing the user query and a generated answer based on relevant documents.

```
{

  "query": "string",

  "answer": "string"

}
```

- **Process**:
  - Downloads necessary files from GCS. Retrieves documents from a Chroma database using similarity search.
  - Ranks and filters documents for relevance using Fine Tuned LLM.
  - Generates a summarized answer using relevant document snippets.

Frontend (Streamlit)

- **Search Interface**: Allows users to input prompts related to social issues.
- **Results Dashboard**: Displays recommended research papers, researchers and insights related to the query.

GitHub Repository Structure

```
.
├── .github/
│   └── workflows/
│       └── pre-commit.yml        # GitHub Actions workflow for pre-commit hooks
├── notebooks/
│   ├── .gitkeep              # Placeholder to keep notebooks folder in version control
│   └── eda.ipynb              # Exploratory Data Analysis notebook
├── references/
```

```
│   ├── .gitkeep              # Placeholder for references folder
│   ├── Flowchart.jpeg           # Project flowchart image
│   └── UI.jpeg               # User interface design image
├── reports/
│   ├── .gitkeep              # Placeholder for reports folder
│   ├── Design_Document.pdf    # Project milestone report
│   └──Test_Document.pdf       # Project milestone report
├── src/
│   ├── api-service/           # API service module
│   ├── api/                 # Main API structure
│   │   ├── routers/           # FastAPI router modules
│   │   │   └── llm_rag_chat.py
│   │   └── utils/                    # Utility modules for API logic
│   │       └── llm_rag_utils.py
│   ├── service.py                   # Main service entry file for FastAPI application
├── Dockerfile                       # Dockerfile for API containerization
├── Pipfile                          # Pipfile for Python dependencies (pipenv)
├── Pipfile.lock                     # Lock file for pipenv dependencies
├── docker-entrypoint.sh             # Entrypoint script for Docker container
├── docker-shell.sh                  # Shell script for Docker operations
├── embed_papers/                    # Scripts for embedding research papers
├── finetuning/                      # Folder for model finetuning scripts and files
├── frontend_ui/                     # Minimal frontend implementation
├── perform_rag/                     # Folder for RAG (Retrieve and Generate) functionality
├── retrieve_papers/                 # Folder for retrieving research papers
├── docker-compose.sh                # Script for Docker Compose operations
├── tests/                           # Testing suite
│   ├── test_embed_papers.py                     # Unit tests for embedding papers
│   ├── test_integration_embed_retrieve.py        # Integration tests for embedding and retrieval
│   ├── test_perform_rag.py                      # Unit tests for RAG functionality
│   ├── test_retrieve_papers.py                  # Unit tests for paper retrieval
│   └── test_app.py                              # System tests
├── .gitignore                       # Git ignore file
├── .pre-commit-config.yaml          # Configuration file for pre-commit hooks
├── LICENSE                          # License for the project
├── README.md                        # Project README file
├── pytest.ini                       # Configuration file for pytest
├── requirements.txt                 # Additional requirements for Python dependencies
└── test_output.tar                  # Test output archive
```

5. Continuous Integration Setup

The CI pipeline, implemented via **GitHub Actions**, triggers on every push or merge event, ensuring the application’s code integrity and functionality.

CI Pipeline Components

- **Code Build and Linting**: We have an automatic build process of the code and utilize Black for Python to maintain code quality.
- **Automated Testing**: Runs unit, integration, and system tests with output reports.
- **Coverage Reporting**: Monitors test coverage, ensuring it remains above 50%.