Generative AI Project Ideas for Graduate Students

This document outlines potential projects for your generative AI course. These are designed to be completed by teams of 1-3 within a one-month timeframe. The projects are categorized by team size and ambition level to accommodate varying levels of experience and interest.

The projects ideas below are for your guidance. You can pick other ideas to work on. Also, a 'more' ambitious idea will not necessarily get you better project scores. We will judge each project proportional to its difficulty. For example, if you pick an easier project, you will be expected to be more thorough and complete in your evaluations.

For each project we will expect you to provide a documented GitHub repository, a short report and a short (<5 minute) demo video showing your system or project in operation. These are also great assets to have if you want to get ahead in the job or internship market.

For both 2- and 3-person teams, you are **required** to have a working website showcasing your project. We are giving you a lot of freedom in how you create this website and showcase your project. You can launch free websites from many places today, including github.io or wordpress etc.

Deliverables/timeline (for all teams):

Oct. 31: Start a blank 'team' project document in GRAIL and share with the three of us. In the project document, list your team name and team members. Give us a (roughly) 1-page summary of your idea. Try to be as specific as possible.

Nov. 15: mid-project check in. We will expect you to provide a few paragraphs on: is everything going as planned or have you pivoted from your idea (if you have pivoted strongly from your idea, then you should come talk to us before this date)? If there is more than one team member, who is doing what?

Dec. 1: The full project report is due by midnight. We expect a detailed report, the format of which we will release by Nov. 15. The demo video and GitHub repo links (as well as the website, if applicable) should be in your project report i.e., nothing should be submitted through Brightspace.

We will start grading the projects right after Dec. 1, so you must not modify anything after that.

One-Person Teams

Project 1.1: Fine-Tune a Small Model on a Niche Dataset

- **Concept:** Take a small, open-source model (like GPT-2 or a DistilBERT variant) and fine-tune it on a specific, niche corpus. This could be the works of a particular author, a set of legal documents, your own creative writing, or a collection of specialized technical articles. You are also free to use cloud services for this, or try a larger model. You could also consider using more than one model or fine-tuning exercise.
- **Goal:** Gain hands-on experience with the fine-tuning process. You will need to prepare a dataset, run the training process, and write a short report evaluating how the model's performance and "personality" change after training.
- **Suggested Steps:**

1. Select and clean a dataset (>1MB of plain text). Do **not** use a pre-prepared dataset that has already been labeled or cleaned!
2. Set up an environment to run the fine-tuning script (e.g., using Google Colab with a free GPU).
3. Use a library like *Hugging Face Transformers* to fine-tune a pre-trained model on your dataset.
4. Test the original and fine-tuned models with the same prompts to compare their outputs.
5. Write a brief report on your findings.

- **Suggested Tools:** Hugging Face Transformers library, Google Colab, a small model from the Hugging Face Hub (e.g., `gpt2, distilbert-base-uncased).`

Project 1.2: Smart FAQ Generator from Documents

- **Concept:** Create a tool that ingests a document (like a course syllabus or a technical manual) and automatically generates a list of Frequently Asked Questions (FAQs) and their corresponding answers. The core of the project involves using a large language model to create relevant question-answer pairs from the text.
- **Goal:** Develop skills in prompt engineering, text processing, and API integration. This project will provide experience in using LLMs for practical information extraction and structuring tasks.
- **Suggested Steps:**

1. Write a script to read and parse a source document (e.g., a `.txt`, `.md`, or even `.pdf` file).
2. Break the document text into smaller, manageable chunks that can be fed into an LLM prompt.
3. Design a prompt that instructs an LLM (like GPT-3.5 or a model from Hugging Face) to generate a question-and-answer pair based on a given text chunk.
4. Process the entire document by feeding each chunk to the LLM and collecting the generated FAQs.
5. Create a simple interface to display the results, either as a formatted text file, a Markdown document, or a basic HTML page.
6. You should try this for at least 3-4 documents/use-cases, preferrably at least a few tens of pages each.

- **Suggested Tools:** Python, a large language model API (e.g., OpenAI, Anthropic, or Hugging Face), text processing libraries (`PyPDF2` for PDFs, `BeautifulSoup` for HTML).

Project 1.3: Personal RAG-Based Q&A System

- **Concept:** Implement a Retrieval-Augmented Generation (RAG) system that can answer questions about a specific, personal set of documents. This could be your course notes, a favorite novel, or a few dozen research papers on a topic of interest. We will cover RAG in the later part of October, but you can already start reading about it if you're interested in this direction.
- **Goal**: Build a complete, functioning RAG pipeline from scratch. This involves document loading, text splitting, embedding generation, vector storage, retrieval, and final answer generation using an LLM.
- **Suggested Steps:**

1. Gather a small corpus of documents (e.g., 5-10 PDF or text files).
2. Write a script to load and split the documents into manageable chunks.
3. Use an embedding model (via OpenRouter or another service) to create vector representations for each chunk.
4. Store these vectors in a simple vector database (like FAISS or ChromaDB).
5. Create a user interface (a simple command-line interface is sufficient) that takes a question, embeds it, finds the most relevant document chunks from your vector store, and passes them along with the question to an LLM to generate an answer.

- **Suggested Tools:** LangChain or LlamaIndex libraries, a vector store (FAISS, ChromaDB), an LLM API (like OpenRouter).

Part 2: Two-Person Teams

Project 2.1: LLM-Powered Code Reviewer

- **Concept:** Create a simple tool where a user can submit a block of code. The tool sends the code to an LLM with a carefully engineered prompt to check for bugs, suggest style improvements, and generate documentation.
- **Goal:** Build a functional code assistance tool that demonstrates strong prompt engineering skills.

  Suggested Steps:

1. Develop a simple UI (can be a basic web page or a desktop app) that accepts code input.
2. Engineer a detailed "system prompt" that instructs the LLM to act as an expert code reviewer.
3. The backend should take the user's code, combine it with your prompt, and send it to an LLM.
4. Display the LLM's feedback to the user in a clear, readable format.

- **Suggested Team Roles:**

Frontend/UI: One person handles the user interface and input/output display.

Backend/Prompt Engineering: The other person focuses on the backend logic, prompt design, and API integration.

- **Suggested Tools:** A web framework like Flask or FastAPI, or a simple GUI library like Tkinter. Any powerful LLM API (OpenRouter is a good choice).

Project 2.2: "Chat with Your Data" Application

- **Concept:** Develop a tool where a user can upload a CSV file and ask natural language questions about it (e.g., "What is the average value in the sales column?"). The tool uses an LLM to translate the natural language question into a data query (e.g., Python Pandas code), executes it, and returns the answer.
- **Goal:** Create a tool that bridges the gap between natural language and data analysis code.

Suggested Steps:

1. Build a UI that allows a user to upload a CSV file.
2. When a user asks a question, send the question and the CSV headers to an LLM.
3. Prompt the LLM to generate Python Pandas code that can answer the question.
4. Crucially: Execute the generated code in a sandboxed environment to get the result.
5. Return the result to the user. For an extra challenge, pass the result back to the LLM to be presented in a more natural, conversational way.

Suggested Team Roles:

- Data/Backend: One person manages the data processing, LLM-driven code generation, and sandboxed code execution.
- Frontend/UI: The other person builds the user interface for file uploads and the conversational chat display.
- **Suggested Tools:** Pandas library, a sandboxing tool for code execution (e.g., a restricted Docker container), Flask/FastAPI for the backend.

Project 2.3: Personalized News Summarizer

- **Concept:** Build a web application that allows a user to input URLs of their favorite news sources, scrapes the latest articles, summarizes them using a pre-trained model, and displays the summaries in a clean, easy-to-read interface.
- **Goal:** This project focuses on the practical application of summarization models and the fundamentals of front-end/back-end architecture. It will provide experience with web scraping, API integration, and creating a simple, user-facing application.
- **Role Breakdown & Suggested Steps:**
    - **Person 1 (Back-End Engineer):**
        1. Develop a Python script using libraries like `requests` and `BeautifulSoup` to scrape the full text of articles from given URLs.
        2. Integrate a pre-trained summarization model (like PEGASUS or T5 from Hugging Face) to condense the scraped article text.
        3. Set up a simple API endpoint using Flask or FastAPI that accepts a list of news source URLs, triggers the scraping and summarizing process, and returns the summarized articles.
    - **Person 2 (Front-End Engineer):**
        1. Design a simple web interface with an input field for users to enter news source URLs.

2. Write front-end JavaScript to send the list of URLs to the back-end API.
3. Create a display area on the page to neatly render the article titles and their generated summaries received from the back-end.

- **Suggested Tools:** Python, Flask/FastAPI, `requests`, `BeautifulSoup`, Hugging Face Transformers library, HTML/CSS, JavaScript.

Part 3: Three-Person Teams

Project 3.1: AI-Powered Presentation Generator

- **Concept:** Build a tool that takes a topic, an article, or a block of text as input and automatically generates a slideshow presentation. Each slide should have a title, bullet points, and perhaps even speaker notes.
- **Goal:** Demonstrate a multi-step pipeline where an LLM is used sequentially for different generation tasks (outlining, summarizing, generating slide content).
- **Suggested Steps:**

1. The first LLM call takes the source text and generates a high-level outline for a presentation (e.g., a list of slide titles).
2. For each slide title in the outline, a second LLM call is made to generate the content (bullet points) for that slide, based on the source text.
3. A third (optional) call could generate speaker notes for each slide.
4. The final output is formatted into a simple presentation format (e.g., Markdown, HTML, or JSON).

- **Suggested Team Roles:**

1. Backend/Workflow: Handles the overall logic, text processing, and chaining the LLM calls together.
2. Content/Prompt Engineering: Focuses on crafting and refining the specific prompts for each stage (outlining, slide generation, notes generation).
3. Frontend/UI: Creates the user interface for inputting the topic and displaying the final, formatted presentation.

- **Suggested Tools:** A web framework, LLM APIs. Consider a library to convert the output to a specific format (e.g., `markdown-to-pptx`).

Project 3.2: Simple Multi-Agent System for Problem Solving

- **Concept:** Design a system of two or three specialized AI "agents" that collaborate to perform a complex task, such as planning an event or outlining a research paper. The agents "talk" to each other by passing text outputs back and forth. For example, a "Researcher" agent could gather information, a "Planner" could structure it, and a "Writer" could compose the final text.
- **Goal:** Understand the principles of multi-agent systems and how to orchestrate them. The core challenge is designing the workflow and the distinct "persona" and instructions for each agent.
- **Suggested Steps:**

1. Define a specific, multi-step problem to solve.
2. Design the roles and responsibilities for each agent in the system.
3. Develop highly-specialized prompts for each agent, defining their persona and task.
4. Create a "coordinator" or "main script" that manages the workflow, passing the output of one agent as the input to the next.
5. Assemble the final output from the last agent in the chain.

- **Suggested Team Roles:**

1. Coordinator/Architect: Designs the overall workflow, determines agent roles, and writes the main script that controls the process.
2. Agent Developer 1 (e.g., Researcher): Implements and perfects the prompt for the first agent in the chain.
3. Agent Developer 2 (e.g., Planner/Writer): Implements and perfects the prompt for the subsequent agent(s).

- **Suggested Tools:** Python or another scripting language, LLM APIs. Frameworks like LangChain or Autogen can be used but are not required; building the orchestration logic from scratch is a key part of the learning experience.