Virginia Tech

Department of Computer Science

Blacksburg, VA 24061

CS 4624 - Multimedia, Hypertext, and Information Access

Spring 2025

Optimizing LLMs for Higher Education

Janna Helvey, Sahil Mhatre, Sahaj Singh, Anthony Marraccini, Ayaan Sheikh

Client: Nick York

Professor: Mohamed Farag

May 1, 2025

Table of Contents

Abstract

The objective of this project is to tailor large language models (LLMs) to higher education for the ultimate aim of expanding access to educational materials for marginalized communities. Around the world, many potential students are excluded from higher education by socioeconomic or geographic barriers. We are building on the past work of our client, Nick York, who did research on "Broadening the Availability of Computer Science Education to Underrepresented Groups Using AI." We are building on this vision by incorporating a Retrieval-Augmented Generation (RAG) model that enables LLMs to leverage external documents—such as quizzes and PDFs—dynamically as contextual input. This enables the model to produce more precise, relevant, and meaningful educational responses. We are evaluating the effectiveness of models such as Mistral-7B, OpenAI's GPT-4.0 and Meta's Tiny LLaMa, actively working on prompt engineering techniques and multi-model validation techniques. We have established a workable RAG pipeline with different models, and have refined prompt design to better facilitate ongoing, high-level educational discussion and personalized learning feedback.

Introduction

Group 18 - Optimizing LLMs For Higher Education.ipynb

2.1 Overview

Access to high-level education remains a global concern, particularly for underrepresented groups. Socioeconomic differences, poor infrastructure, and geographical isolation prevent most students from accessing advanced educational resources. With the increasing promise of large language models (LLMs), there is a chance to bridge this educational gap using AI-based tools. However, contemporary LLMs are faced with a number of challenges which hinder their effectiveness in educational settings, such as a lack of context understanding, intermittent generation of misleading information, and incapacity to generate consistently good quality educational conversation.

2.2 Educational Accessibility

Many capable students are excluded from the mainstream education pathways due to factors beyond their control. These include economic limitations, lack of access to institutions, or unsafe environments. The overall aim of this project is to continue the democratization of education by improving the reliability and context-awareness of LLMs to serve as intelligent learning companions. By providing an AI-driven system to support sophisticated learning, we are seeking to make better available the precious educational opportunity for these excluded groups.

2.3 Technical Limitations of LLMs

Although models like GPT-4.0 andTiny LLaMa have excellent language generation skills, they are not necessarily intended to provide accurate, contextualized answers for high-stakes education. Without the domain of context, they default to generalized or false answers. This deficiency is particularly egregious in education, where fact accuracy and relevance are paramount. LLMs also seldom retain context across multiple turns, which diminishes their utility in longer educational conversation.

2.4 Integration Challenges

To address the limitations of generic LLM responses, our team has developed a Retrieval-Augmented Generation (RAG) system enabling the model to learn from external sources like PDFs and quizzes. While useful as this improves accuracy, introducing RAG in full force generates issues in prompt creation, context establishment, and system tuning. Our focus is now on enhancing these elements through targeted experimentation and backend optimization

2.5 Building on Previous Work

This project builds upon our client, Nick York's, prior research on leveraging AI to broaden access to computer science education for underrepresented groups. Extending from this foundation, we aim to further enhance the educational utility of LLMs and experiment with methods that can be applied in real learning environments. Our priorities are the technical polishing of the backend processes instead of front-end usability at this stage, with a focus on the core functionality and research feasibility of the system.

Requirements

Our goal was to test many different approaches, including different LLMs, RAG and prompt engineering techniques, meaning we needed to find an application that could easily store previous attempts, while still running current updated code. We wanted an intuitive interface that can be used by researchers to continue our work on optimizing techniques and implement our findings into their own work.

3.1 User Requirements

The system we decided to use was Google CoLab. As this project is based on researching and implementing different techniques for RAG, prompt engineering, and LLMs, we identified only a single user using this system, researchers. In the future of the project, there will also be Admins and Student users, but currently, all development is focused on the expectations and usage of a researcher.

3.1.1 Researcher User Needs

Researchers already have an understanding of the techniques we are using and understand the nuances of using an LLM. While some researchers may have programming skills, others may not, so providing in depth comments in our code will ensure that all researchers can access and understand our developments.

Main considerations for researchers:

- Keep previous implementations
- Provide the source of our training data
- Explanation of choices

3.2 Technical Requirements

To ensure that the system operates reliably, we have outlined specific technical criteria. The CoLab itself requires an **up to date browser and a google account.**

3.2.1 Performance Requirements

Our CoLab will run medium to large LLMs. This means that there will be a large amount of RAM used. It needs to process large amounts of data, vectorize it, and use an LLM to finally answer specific questions based on the previously stored information. Our requirements set out necessary choices to make sure the LLM is able to respond to a user's query, in an efficient manner.

- Users must be connected to the CoLab's GPU
- Users must run all commands on each iteration of the testing

3.3 Functional Requirements

The system must provide specific functionalities:

3.3.1 Core Features

- File Upload
    - Accept PDF documents to use as training data
    - File validation
    - Path checking
- Model Upload
    - Use an uploaded model as the chosen LLM
- Read And Split Data

    - Iterate through a list of file paths and load text content from file
    - Split text into vectorized chunks
- Retrieve Data
    - Retrieve relevant data from database
- Define LLM
    - Define the Parameters of the LLM
    - Load the LLM into the session
- Memory
    - Store conversations in memory as context for future responses
- Use the LLM
    - Define the RAG pipeline
    - Define the conversation template
    - Respond to student queries
- User Quiz
    - Ask the user questions for background context
    - Feed context to LLM
    - Recommend an LLM template for students
- Student Quiz
    - Test student knowledge before and after use of the LLM

3.4 Conclusion

These requirements ensure that our application transforms a simple user query into an optimal LLM response. The focus is on research techniques, with all previous implementations being stored within the same CoLab, to better understand our process and what changes we implemented. Key considerations throughout the requirements:

- Accessibility for non-technical researchers
- File upload capabilities
- LLM functionality and efficiency

Design

The system we designed is built as a Retrieval Augmented Generation system utilizing the LangChain Framework. This architecture allows the system to combine large language models (LLMs) with the ability to utilize external knowledge sources. The following are the core components of the system.

4.1 Document Loading

The system uses document loaders such as PyPDFLoader and Docx2txtLoader to extract text context from various file formats including PDF, DOCX, and .txt. This is followed by a RecursiveCharacterTextSplitter which processes the extracted text into chunks, assisting in optimizing the LLM for efficient retrieval.

4.2 Web Crawling

The web crawler that implemented crawled over the OpenDSA website and extracted the texts from the available books. This allowed the system to access large amounts of data from a trusted source, but allows a user to change the link to a source of their choice.

4.3 Knowledge Representation and Storage:

The system uses HuggingFaceEmbeddings to generate embeddings and uses a Vector Database (Chroma) to store and index the embeddings for further efficient retrieval.

4.4 LLM

Our system uses Mistral-7B LLM. It is loaded in with carefully set parameters which are optimized for our task. We maximize GPU utilization by offloading layers (setting gpu layers to -1). We also set the batch size to 512 as this was the most optimal value we tested. Other tweaked parameters which affect our systems performance include temperature, max_tokens, and top_p.

4.4.1 Retriever

Our retriever, Chroma, helps facilitate the retrieval of relevant text chunks based on the user input. Chroma searches for semantically similar text in our vector database.

4.4.1 Memory

To further improve our systems accessibility, we have implemented conversational memory. Using ConversationBufferMemory, we maintain a history of previous interactions which allows us to preserve context across multiple interactions.

4.4.1 Orchestration and Execution

Most of the preceding components are linked together using LangChain's chain functionality which allows us to form a cohesive workflow and present the user with a relevant LLM-generated response.

4.5 Quiz Design

To measure the efficacy of our system, we added a quiz module that assesses a student's understanding of basic computer science concepts prior to and following exposure to the LLM. The module supports formative and summative assessment and is fully integrated into our RAG-based learning pipeline.

The system generates quizzes from either the LLM itself or a fallback set of questions to use in case of failure. When content is asked from the vector store, a structured prompt is given to the LLM to request it to generate a multiple-choice question from a given document chunk. Each question generated has to be in a strict format: four choices (A–D) and a clear marking of the correct choice. For stability, a regex-based parser extracts the question, choices, and answer from the output of the LLM. If parsing fails or if there is a lack of content, the system selects a validated default question from a carefully curated question bank with data structures and algorithms.

Each quiz contains three questions. There is a pre-interaction quiz given before the LLM is used to establish a knowledge baseline, and a post-interaction quiz after to assess gain. After each quiz, answers are recorded and scores calculated and saved. Gain is then reported by comparing the pre and post scores and highlighting areas of improvement or concern. An analysis at the question level is also given to indicate the most difficult concepts.

For each quiz session, results are saved as dated JSON files, allowing instructors or researchers to track performance over time. The quiz module is capable of execution in a stand-alone mode or in conjunction with interactive LLM sessions. The module can also be made more flexible and include other capabilities such as future support for features like adaptive difficulty, student profiling, and specific study guide generation based on incorrect answers.

Implementation

5.1 Frontend Development

After discussion with our client, Nick York, we collectively decided that the front end is not a necessary deliverable for this project. Instead, the deliverable changed to be a Google CoLab, with sufficient backend functionality, and printable responses, acting as our frontend.

5.2 Backend Development

5.2.1 Testing Data and Testing Data Retrieval

At the moment, current testing data is provided in a PDF uploaded by the user. This document was hand retrieved, and has limited information, a single topic. Efforts are being made to make this an automated process, allowing for more in depth information from a reputable source, OpenDSA. As we continue this project, we plan to make a web crawler to crawl the OpenDSA website and store the information in a PDF that will automatically be loaded as context for the LLM.

5.2.2 Libraries

- Langchain: A framework for building applications with language models
- Operator: Extracting items from iterable
- langchain_huggingface: A LangChain integration for Hugging Face embeddings
- Huggingface_hub: Enables downloading models and data from the Hugging Face Hub
- Google.colab: Google Colab-specific utilities
- Termcolor: Adds color to terminal output

- Os and Sys: Interacting with the operating system and system functions
- Chromadb: A vector database used for storing and querying embeddings
- Tiktoken: A tokenizer
- Pypdf: A library for handling PDF files
- Docx2txt: Extracts text from DOCX files
- Llama-ccp-python: A Python binding for Llama models, enabling local inference
- Sentence_transformers: Provides sentence embeddings for NLP applications
- Ipywidgets: Enables interactive widgets in CoLab
- Nvidia-smi: Checks the GPU status in a system

5.2.3 LLM Definition and Usage

Since our frontend is the output of the Google CoLab, the LLM currently has no chatbot UI or direct user interaction besides the prompt terminal, therefore, we are considering the LLM definition and usage to be backend work.

We were able to follow a well tested template for interaction with the LLM, meaning that our current starting prompt template is the baseline for our testing and future improvements to the prompt, as part of our prompt engineering efforts.

The quiz functionality allows for a “Chatbot” like feature

Similarly, at this time we have a baseline for:

- LLM running time
- LLM accuracy
- LLM response helpfulness (objective)

This allows us to use our current backend LLM implementation as a baseline for the techniques we will be using for the remainder of our project.

User’s Manual

This manual guides users through operating the Optimizing LLMs for Higher Education Google CoLab system. It describes the current setup and provides an overview of planned future enhancements to further improve usability and functionality.

6.1. Overview

The system is implemented as an interactive Google CoLab notebook that integrates a Retrieval-Augmented Generation (RAG) pipeline with large language models (LLMs). Currently, the notebook allows researchers and educators to upload documents and generate responses. Looking ahead, our team plans to expand features based on user feedback and evolving requirements. The system supports uploading PDF documents as context sources and includes a basic web crawler to extract educational content (e.g., from OpenDSA) and convert it into a usable format (PDF) for the LLM.

6.2 System Requirements

Before using the system, ensure you have:

- **Hardware:** A computer with stable internet access.
- **Software:** A modern web browser (Chrome, Firefox, Edge) with JavaScript enabled.
- **Accounts:** A valid Google account for accessing Google CoLab.
- **Resources:** A GPU-enabled CoLab runtime for efficient processing of large LLMs.

6.3 Getting Started

6.3.1 Accessing the Notebook

Open the Google CoLab link using the provided URL.

Confirm you are signed in with your Google account.

Enable GPU by navigating to **Runtime > Change runtime type** and selecting "GPU."

Run the initial setup cells:

- **Block 1:** Installs necessary libraries (langchain, llama-cpp-python, etc.).

- **Block 2:** Imports required modules.

- **Block 3:** Enables widget support.

6.3.2 User Interface Overview

Code Cells: Each cell contains code that handles data uploads, preprocessing, and query execution.

Output Display: System outputs appear directly under the executed cells.

Interactive Prompts: Some cells will require user input. Follow on-screen instructions.

6.4. Operating the System

6.4.1 Uploading Files and Data

File Upload:

Use the CoLab “Upload” option to import documents (.pdf, .docx, .txt).

Run **Block 7** to load and parse the file.

Change the input file path in **Block 6** (filepath variable).

Data Validation:

The system automatically checks file format and path existence.

6.4.2 Running the RAG Pipeline

- Preprocessing Data:

  Run **Block 7** to load the file and **Block 8** to split it into vectorized chunks.

  LLM Initialization:

  Run **Block 4** to download and initialize the model (e.g., LLaMa, CapybaraHermes).

  Query Execution:

  Run **Block 26** to prompt the LLM.

  Enter your query into the designated input field inside the cell.

6.4.3 Web Crawling Capabilities

- Crawling Educational Sites:

  Run **Block 28** to scrape OpenDSA or similar websites.

  Change the target URL in **Block 28** (crawl_url variable).

  PDF Conversion:

  Crawled content is converted to PDF and passed through the ingestion pipeline automatically.

6.5 Troubleshooting and FAQs

- GPU Not Enabled:

  Navigate to **Runtime > Change runtime type > Hardware Accelerator > GPU**.

  File Upload Errors:

  Make sure the uploaded file exists, and its format is supported (.pdf, .docx, .txt).

  Web Crawl Issues:

  Verify that the target website is accessible and contains readable text.

6.6 Versions for All Imports

Python 3.10.12 | langchain 0.1.13 | llama-cpp-python 0.2.44 | sentence-transformers 2.2.2 | PyPDF2 3.0.1 | beautifulsoup4 4.12.3 | requests 2.31.0

6.7 Contact and Support

For any issues not covered in this manual, please refer to the Developer Manual. Additional support documentation and troubleshooting guides will be made available.

6.8 Feature Usage Guide

This section outlines how to use each key feature, which code blocks must be run, and where to modify inputs if needed.

6.8.1 Uploading and Processing a Document

Purpose: Upload a file (PDF, DOCX, or TXT) and prepare it for use with the LLM.

Required Blocks:

- **Block 7:** Loads the file.

- **Block 8:** Preprocesses and splits text into chunks.

Where to Change Input:

In **Block 6**, update the filepath variable with your uploaded file’s path (e.g., filepath = "/content/your_file.pdf").

6.8.2 Querying the LLM

Purpose: Ask a question and receive an LLM-generated response using your uploaded document as context.

Required Blocks:

- **Block 4:** Loads and initializes the language model.

- **Block 26:** Accepts user input and displays the LLM's response.

Where to Change Input:

In **Block 26**, enter your custom question in the input() prompt or variable assignment.

6.8.3 Web Crawling Educational Content (OpenDSA)

Purpose: Automatically crawl educational sites and convert content into a PDF for use in the RAG pipeline.

Required Blocks:

- **Block 28:** Crawls the web and saves the content.

- Followed by **Block 7** and **Block 8** to process the generated PDF.

Where to Change Input:

In **Block 28**, modify the crawl_url variable to target a different educational page (e.g., crawl_url = "https://opendsa.cs.vt.edu").

6.8.4 Changing the LLM or Model Parameters

Purpose: Use a different LLM model or tweak its configuration.

Required Blocks:

- **Block 4**

Where to Change Input:

In **Block 4**, edit the model path or configuration (e.g., model name, number of threads, temperature).

Developer’s Manual

7.1 Code Blocks

At the moment, we believe our only users will be developers, and that our developer manual would be the same as the user manual. However, a developer must know the usage and importance of each code block:

Block 1 (updated): Installs necessary packages for LangChain, LlamaCpp, sentence-transformers, document processing **plus the new crawler dependencies** (beautifulsoup4, aiohttp, lxml, langchain_text_splitters).

Block 2: Imports necessary modules and libraries for document loading, text splitting, embeddings, vector storage, and other functionalities.

Block 3: Enables custom widget manager in Colab.

Block 4: Downloads and sets up the CapybaraHermes-2.5-Mistral-7B-GGUF language model from Hugging Face.

Block 5: Imports necessary modules for document loading and text splitting.

Block 6: Specifies the path to the document to be processed and checks if it exists.

Block 7: Loads text content from the specified document (PDF, DOC, or DOCX).

Block 8: Splits the loaded text content into smaller chunks using a RecursiveCharacterTextSplitter.

Block 9: Imports necessary modules for embeddings and vector storage.

Block 10: Initializes HuggingFaceEmbeddings for generating text embeddings.

Block 11: Enables custom widget manager (repeated).

Block 12: Creates or loads a Chroma vector database and stores the text chunks with their embeddings.

Block 13: Tests the retriever by querying the database and printing the content of the first retrieved document.

Block 14: Imports necessary modules for LLM interaction, prompt templates, callbacks, and memory.

Block 15: Defines a custom callback handler to display colored output during LLM token generation.

Block 16: Defines a function to load the LlamaCpp language model with specified parameters.

Block 17: Checks if the model file exists and prints a message accordingly.

Block 18: Displays documentation for the LlamaCpp class using the ? operator.

Block 19: Sets up a ConversationBufferMemory to store conversation history.

Block 20: Defines a prompt template for interacting with the LLM in a conversational context.

Block 21: Imports necessary modules for chain loading and runnable components.

Block 22: Defines a function to load a retrieval-augmented generation (RAG) chain.

Block 23: Defines a function to load a chain without retrieval augmentation.

Block 24: Deletes the existing llm object if it exists.

Block 25: Reloads the LLM and tests it with a simple prompt.

Block 26: Initializes and uses the RAG chain for interactive question answering.

Block 27: Initializes and uses the chain without retrieval augmentation for interactive question answering.

7.2 New blocks for web-crawling & HTML processing

Block 28: Implements a depth-controlled web crawler that fetches course content (e.g., OpenDSA). Uses aiohttp + BeautifulSoup to download HTML, respects robots.txt, and saves raw pages to /content/crawled/.

Block 29: Runs an HTMLHeaderTextSplitter followed by a secondary RecursiveCharacterTextSplitter to break crawled HTML into ~1 000-token chunks with 30-token overlap.

Block 30: Sanitizes metadata on each chunk (converts non-string values to strings) to prevent Chroma ingestion errors.

Block 31: Persists the crawled chunks and their embeddings into the existing Chroma vector store (or creates it if missing).

Block 32: Prints each chunk's metadata["source"] path/URL to verify crawler coverage and assist debugging.

Block 33: A basic testing LLM invocation

Block 34: This is an initial quick quiz to help the LLM decide how to answer question, using years of experience, languages they know (if they ask programming questions) if they have formally studied CS and what topics interest them

Block 35: Beginner LLM, this is made for beginner students and has added context to reply in a meaningful way for a new student

Block 36: Intermediate LLM, this is made for intermediate students, includes a system instruction to respond at a higher level

Block 37: Different test LLM invocation cell

Block 38: Quiz based learning assessment

Testing/Evaluation Assessment

8.1 Testing/Evaluation/Assessment

Our evaluation strategy focuses on ensuring the reliability of quiz processing, the accuracy of incorrect answer identification, and the effective invocation of our LLM. With the application developed across a Colab notebook, testing is performed through interactive cells, manual execution, and controlled experiments.

8.2 Notebook-Based Functionality Testing

Without a dedicated web front end, we validate functionality using Colab notebooks:

- **Simple Interaction Submission and Response Processing:**
  Notebooks are used to simulate user experience. LLM invocation cells include input and response, meaning manual verification is required so that incorrect answers are correctly flagged.

- **Quiz Interaction Submission and Response Processing:**

  The user interacts with the quiz and is automatically graded. The LLM creates and grades the quiz, meaning no user testing is necessary besides verifying that the quiz questions and answers are correct. Currently, the quiz has been accurate and no anomalies have been flagged.

8.2 Testing Steps:

- Run the CoLab from the beginning
- When LLM invocation occurs, look for incorrect information, phrasing, or comparison

8.3 LLM Integration and Training Data Testing

Informed by our high-level LLM project, our testing of the LLM integration covers:

8.3.1 Training Data Consistency:

We have many sources of context for the LLM. This can include, but is not limited to: a user uploaded PDF, vectorized data from crawled web links, the individual student quiz, and the background Q&A for each level of LLM. This means that while each LLM call will have different levels of context, we have found that the more we can provide, the better and more tailored our responses will be.

8.3.2 Data Retrieval Validation:

Test cells include predefined queries, and the outputs are compared against expected contextual information.

8.3.3 LLM Invocation Parameters:

Testing cells ensure that the model is initialized with appropriate parameters, such as a context length optimized to balance efficiency and detail (e.g., n_ctx_per_seq is maintained below n_ctx_train).

8.3.4 Student Quiz:

Currently, the quiz has fallback responses if there is a need for it, but is otherwise fully graded and assessed by the LLM. Visual check to make sure it is correct is required.

Lessons Learned

9.1 Timeline/Schedule

Date	Milestone Description
2/6	Reached out to Nick for client meeting, scheduled and began looking over previous work
2/13	Background research and chose optimal LLM model
2/24	Begin initial integration of chosen Hugging Face Models
3/17	Establish functional RAG pipeline, Conduct initial accuracy testing
3/24	Refine and optimize model from tests, Modify RAG to increase context
3/31	Conduct a mid-project review, Prepare early stage demos showing functionality
4/8	Check in with Nick, look for guidance on current implementation and steps to move forward with experimentation
4/17	Fix bugs and cleanup before final presentation

9.2 Problems

There were many issues that arose with our work on this project. However, we were able to identify and tackle many of these issues head on and create relevant solutions.

9.2.1 Problem 1: Hallucinations

The first problem that we ran into was the LLMs returning untrue information. A standard question of, “What is the time complexity of bubble sort?” was asked to the LLM. When it was returned that bubble sort was the most commonly used sorting algorithm, with a time complexity of O(log(n)), we realized that something was incorrect. Our first choice was to add more context, with more information for the LLM to learn from before answering. This was given in the “Background Q&A”, seen in the LLM invocation templates. This was identified by Nick as one of his main issues, so we worked quickly to remedy this.

9.2.2 Problem 2 - Repeating Answers for “What vs Why” questions

This issue was identified as an issue when asking questions such as “What is insertion sort?”, wait for response and follow up with “Why would you use insertion sort?”. This typically led the LLM to respond with the exact same answer, even when the questions were different. This was fixed by ensuring that the data from the crawled websites was properly stored in the database and changing our model. At this point, we had been looking to improve our speed, and chosen to work with Tiny-Llama, in an effort to improve response time speed. However, this took a detrimental downturn to our response accuracy, and therefore, we switched our model back to Mistral-7B. With the correct pipeline in place, Mistral was able to correctly answer ‘What vs Why’ questions, with examples as needed.

9.2.3 Problem 3 - Response Time

One of our biggest challenges, one that is still ongoing, is the LLM response time. What we found is that questions can take up to 15 minutes for a single response. We discussed the possibility of testing different LLMs with our client and decided to try using a small LLM with less parameters to get a quicker response. This did work, as Tiny-Llama's response time was typically around 3-5 minutes, around a 75% decrease; however, we noticed a significant decline in answer reliability and clarity, leading us back to Mistral. A second attempt was made, instead opting for a larger, more well known model, in the hopes that OpenAI's ChatGPT-4.o would answer quicker. However, as expected, response time only went up - nearly to 30 minutes, leading us back to Mistral's model. In future work, we hope that this time can be decreased while maintaining our current levels of response quality.

9.2.4 Problem 4 - Lack of Context

As our client had mentioned, we struggled with giving the LLM enough context to provide a personalized response that was still accurate. After speaking with Nick, he suggested multiple templates for different levels of students. Therefore, the LLM could respond with the correct level of jargon without overwhelming a new student. We developed two levels of LLM, beginner and intermediate, with a template to follow for high levels for future reference. As mentioned previously, we added Background Q&A's as a way to give reference examples for the LLM to respond with, and provide more context. However, we also developed a student profile quiz. This quiz asks the current student to answer personal questions, from their interests, years of programming experience, and more. The model uses the information as context for the student, and passes on a recommendation of which LLM to use. Currently, this is hardcoded based on their years of experience, but can be expanded to be an LLM given recommendation in the future.

9.2.4 Problem 4 - GPU Space

The final problem we identified was our lack of GPU space. Running the CoLab all the way through one time, with 4-5 calls to the LLM is enough to burn through the free allocation of GPU usage for users. This means that it is very difficult to test, and in the future, will make it difficult for students to use. It also raises an issue for when the program is moved to a local machine concerning CPU or disk usage. A way to fix this is by paying for more GPU usage, however, we were able to develop our work within this constraint, but recommend this as an option for a future team.

9.3 Future work

- Model Optimization & Multi-Model Integration
    - Benchmark newer LLMs (LLaMa2, GPT-3.5 Turbo) for trade-offs in response quality and time.
- Adaptive Prompt Engineering
    - Fine-tune prompts based on student level (Beginner vs. Intermediate).
    - Introduce context-aware prompt templates with quiz history integration.
- Enhanced Document Processing
    - Expand file format support beyond PDF/DOCX (e.g., HTML, JSON).
    - Improve document parsing for better chunking and metadata extraction.
- Frontend Prototype
    - Build simple UI outside of Colab.
    - Display LLM responses, quiz interface, and performance insights.
- Analytics & Evaluation
    - Add quiz performance tracking dashboards.

- Measure improvement in concept retention pre/post LLM interaction.

Acknowledgments and References:

- **Yusuf Elnady** for the Google CoLab Template (https://colab.research.google.com/drive/1Zs70RvMVBnn4VFUfS-Rst4dG0fojnEaT)
- **Nick York (nickyork@vt.edu)** for his previous work and guidance
    - Reach out to Nick for access to his GitHub repository, where our initial reference code was sourced
- **Mohamed Farag(mmagdy@vt.edu)** for answering any questions we had along the way and providing good insights on how to improve our, and help us get on the right track