UNIVERSITY
OF SUSSEX

# Revision Assistant based on LLM.

Candidate Number: 263156
BSc Computer Science & Artificial Intelligence
School of Engineering and Informatics
Project Supervisor: Prof Julie Weeds
Year of Submission: 2023/2024

## Statement of Originality

This report is submitted as part of the requirement for the degree of Computer Science & Artificial Intelligence at the University of Sussex. It is the product of my own labour except where indicated in the report. The report can be freely copied and distributed, provided the source is acknowledged. I hereby give permission for a copy of this report to be loaned out to students in future years.

Candidate Number: 263156 Signed: Date: May 2, 2024

## Acknowledge

I would like to thank my project supervisor, Prof Julie Weeds, for her support and guidance throughout this project.

# Table of Contents

# 1. Introduction

## 1.1. Project Description

In this project, I developed a revision assistant web application that utilises the Large Language Model (LLM) for Automatic Test Generation (AQG). The core function of the application is to receive books, notes and audio or video files of lectures from students as input. With the help of Natural Language Processing (NLP) technology, the application can analyse this material and generate appropriate Multiple-Choice Questions (MCQs) quizzes based on the content. In addition, it provides personalised feedback based on students' test answers, helping them to understand the reasons for their mistakes and thus consolidate their knowledge more effectively.

I designed this application with the intention of helping students to review better and digest what they have learned in their courses, especially for the Massive Open Online Courses (MOOCs) that are increasingly popular on the Internet nowadays. Through this web application, students can take self-tests anytime and anywhere to keep track of their learning progress and thus improve their learning efficiency. Meanwhile, it helps teachers save time in designing quizzes and promotes the development of MOOCs.

In addition, the project also considers the ability to process a wide range of learning materials, such as text, audio, and video, which can be effectively converted into structured quiz content by the application. This not only makes the learning process more diverse but also allows for a deeper and more comprehensive understanding of knowledge.

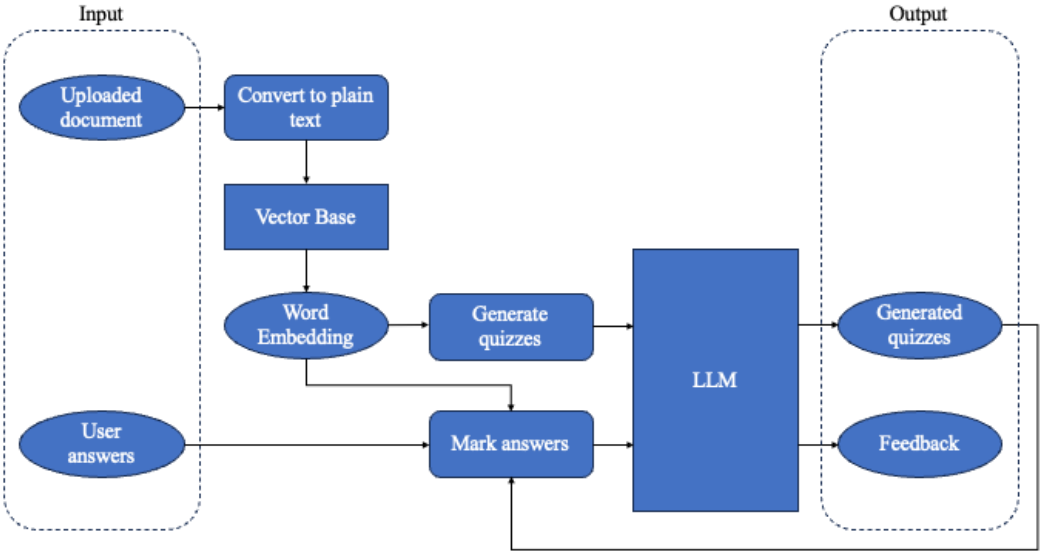



*Figure 1 | High-level Design for this project. The ellipses, straight rectangles and rounded rectangles denote variables, tools and functional requirements for this project, respectively.*

## 1.2. High-level Design

As shown in *Figure 1*, the project fully utilises LLM's excellent text-generation capabilities to generate quizzes and provide feedback. The application relies on two core tools: LLM and a vector database.
The quiz generation phase: at the beginning, the documents uploaded by users are converted to plain text format. The plain text is then converted into word vectors and stored in the vector database. Finally, these word vectors are used to generate targeted quiz questions via LLM to ensure that the questions are closely related to the document's content.
Feedback Generation Phase: After the user completes the quiz and submits their answers, the application generates personalised feedback via LLM based on the user's answers, the content of the generated quiz and the word vectors of the uploaded document. This process takes into account the specifics of the user's answer to provide feedback that helps learning and understanding.
This design aims to improve the efficiency and effectiveness of user learning by providing an interactive and personalised learning environment through an automated quiz and feedback generation process.

## 1.3. Requirements Analysis

### 1.3.1. Functional Requirements

1) This application can accept input in different file formats: text, doc, pdf, audio, and video files.
2) This application can convert docx, PDF and other documents into plain text.
3) This application can transcribe audio into text using the Whisper model.
4) This application can perform word embedding on the text and store it in a vector database.
5) This application can use different prompts to generate MCQ quizzes based on files of various sizes.
6) The application can mark users' answers and give personalised feedback to the user based on uploaded material from the user.

### 1.3.2. Non-Functional Requirements

1) This application should be compatible with mobile and desktop devices.
2) This application should operate in real-time.
3) This application shall use open-source technologies and be accessible to all.

## 1.4. Background Research

### 1.4.1. Use-case of AQG

Traditional AQG methods are mainly rule-based techniques that generate MCQs based on predefined templates and keywords [1]. In addition to this, advanced methods such as Encoder-decoder, Generative Adversarial Networks, Deep Reinforcement Learning, and Transformer [2] have been developed in recent years.

Among these methods, Lakshmi's research [2023] uses the Transformer framework. The system takes text as input and outputs corresponding MCQs. It first uses Bidirectional Encoder Representation from Transformers (BERT) [3] based on Transformer architecture to understand and summarise the text. Then, keywords are extracted from the summarised text using a keyword extraction algorithm. Next, the system locates the key sentences in the summarised text related to the selected keywords through sentence mapping. Finally, it generates disturbance terms for MCQs based on these key sentences using WordNet. [4]

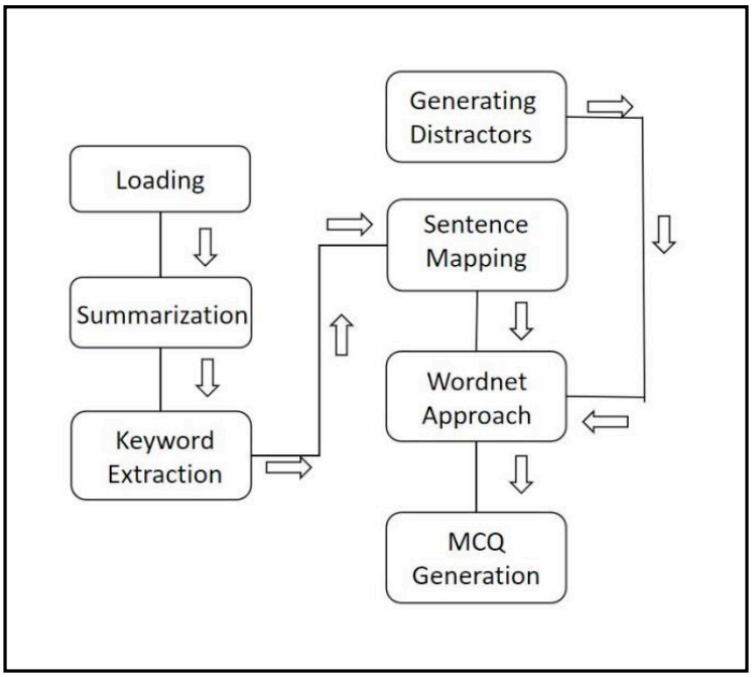



*Figure 2 | Architecture of Lakshmi's [2023] System* [4]

Although the above methods can generate MCQs, the generated questions tend to be in a single format and fail to provide personalised feedback to students. Given that the Transformer technology has demonstrated a better understanding of text and generation capability, I hope to gain a deeper understanding of textual content through LLM. Thereby, I can generate more flexible and diverse MCQs and provide personalised feedback.

### 1.4.2. Tools and Technologies

#### 1.4.2.1. Streamlit

Streamlit[1] is an open-source Python library that enables developers to quickly create and deploy web applications for data science and machine learning projects. Developers can use Streamlit to transform data scripts into interactive web applications in minutes, all in pure Python without front-end development experience. Streamlit provides multiple page elements such as text elements, data elements, chart elements, input widgets, and more. This application uses Streamlit to deploy front-end pages and enable efficient interaction with the back-end. This integration not only improves development efficiency but also optimises the user interface experience.

#### 1.4.2.2. LangChain

LangChain[2] is a high-level framework designed to facilitate the development of LLMs-powered applications. Developers can build applications using open-source building blocks, components, and third-party integrations provided by LangChain. LangChain integrates Document Loader, Text Splitter, Text Embedding Model, Vector Store, and LLMs. The Document Loader loads source data into the Document class. A Document contains a piece of text and associated metadata, and the source data can be a .txt file, a PDF file, a Microsoft Word document and so on. Text Splitters can split long documents into smaller chunks. Text Embedding Models can convert a piece of text into a vector representation. Vector Store can store embedded data and provide functions such as vector searching. The LLMs component integrates standard interfaces for interacting with different LLMs. In addition to this, LangChain also provides Prompt Templates, Chains and other features.

[1] https://docs.streamlit.io/
[2] https://python.langchain.com/docs/get_started/introduction

### 1.4.2.3. Whisper

Whisper [3] is a general-purpose automatic speech recognition system trained on multilingual and multi-task supervised data collected from the web. The Whisper architecture is a simple end-to-end approach implemented as an encoder-decoder Transformer. Using such a large and diverse dataset in Whisper improves robustness to accents, background noise and technical language. In addition, it supports transcription into multiple languages and translation from those languages into English. As shown in *Figure 3*, This model's performance is already close to human-level transcription. [5] In my experiment, the base.en model got a Word Error Rate (WER) of 4.26%, the large-v2 model got a WER of 2.65%, and the large-v3 model got a WER of 2.03% on the LibriSpeech clean test dataset [6]. The Whisper model allows you to convert user uploaded audio or video into plain text.

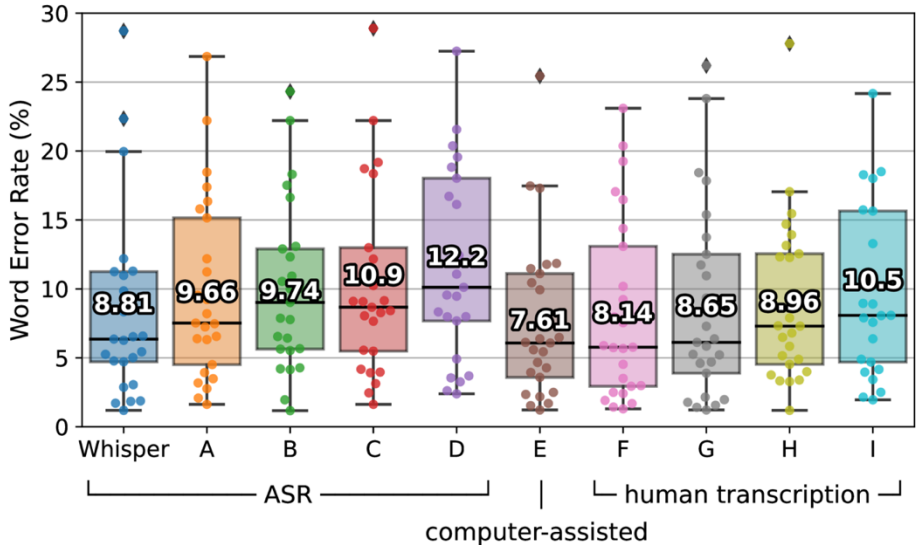



*Figure 3 | Whisper Performance.* [5]

### 1.4.2.4. Chroma

LLMs cannot inherently understand text semantics directly, so we need to map words into vectors in a continuous high-dimensional vector space called embeddings to allow the model to capture the semantic relationships between words. Chroma[4] is a vector database for building AI applications with embeddings. It processes plain text to generate embeddings, which are stored within the database for efficient retrieval. Chroma can be saved and loaded on the local computer. Additionally, it can be combined with LangChain's vector store component to implement vector data storage and retrieval functions easily.

---

[3] https://openai.com/research/whisper
[4] https://docs.trychroma.com/

#### 1.4.2.5. LLMs

LLMs have achieved great success nowadays, and they originate from the Transformer structure [2]. Through the development of Bert [3], Generative Pre-trained Transformer (GPT) series [7], [8], [9] and other models, it fully proves the ability of the Transformer structure on the text understanding and generation. Finally, today's GPT [10], [11], Llama [12], [13] and other LLMs appeared. They are all the results of the stacking of the Transformer structure and the massive dataset training.
GPT3 [9] demonstrates LLMs' strong few-shot or zero-shot capability through prompts. Prompt helps the model to understand and complete the task better by providing contextual information. This enables LLMs to achieve good results without pre-training on a specific task, avoiding the consumption of large amounts of pre-training resources and greatly enhancing the generality of LLMs. This means we can use different prompts to make LLMs perform tasks such as quizzes and feedback generation.

## 1.5. Motivation

Quizzes can help learners assess and consolidate their knowledge, but designing them manually is time-consuming. The rise of MOOCs has made knowledge more accessible but has also increased the need for high-quality, personalised quizzes [14].
The excellent text generation ability of the LLMs allows LLMs to generate richer quizzes easily via prompts [11]. Beyond this, LLMs can provide personalised feedback, bringing MOOCs closer to school learning. The AQG through LLMs not only saves educators a lot of time and effort but also significantly improves students' learning experience. This approach provides immediate feedback on student answers based on the input material. So, LLMs can complement the need for high-quality, personalised education assessment by the huge number of MOOCs.

## 1.6. Relevance

This project is relevant to my BSc Computer Science & Artificial Intelligence degree for several reasons. This application implements a revision assistant through NLP techniques, which I have studied in the Natural Language Engineering and Advanced Natural Language Engineering modules. Software engineering techniques are also fundamental skills for this specialisation. In addition, the projects I have done on sentiment analysis using Bert, text-image retrieval using CLIP and video perception based on LLM are all closely related to this project. Through this project, I will learn more LLM-related skills.

# 2. Professional Considerations

## 2.1. BCS Code of Conduct

The project will strictly follow the BCS Code of Conduct[5] to ensure that professional and ethical standards are met in all aspects. Specifically:

1) Public Interest
   I will carefully protect user-uploaded documents and feedback to ensure user privacy and data security. All relevant literature, organisations and authors will be referenced appropriately.
2) Professional Competence and Integrity
   I have been professionally educated and have experience working on projects related to this project. This project will allow me to expand my knowledge of software engineering and LLM.
3) Duty to Relevant Authority
   All work on this project will be done independently by myself. I will follow the principle of academic integrity and avoid any possible academic misconduct, such as plagiarism or fraud.

## 2.2. Ethical Issues

I have reviewed the Research Ethics Guidelines[6] from the university. The ethical issues we should take care of in this project consisted of using LLM to generate text content based on user-uploaded documents. This project will save documents for each user that they have uploaded, which requires the relationship between the document and the user, but this will not require any personal information about the user (e.g., real name, religion, etc.). To avoid data from being accessed by people not related to the project, all data will be saved on a password-protected laptop and will be deleted at the end of the project. I will ensure participants are kept safe and the utmost care is taken to prevent harm.

[5] https://www.bcs.org/membership-and-registrations/become-a-member/bcs-code-of-conduct
[6] http://www.sussex.ac.uk/staff/research/governance/checklist

# 3. Implementation

## 3.1. Front-End Development

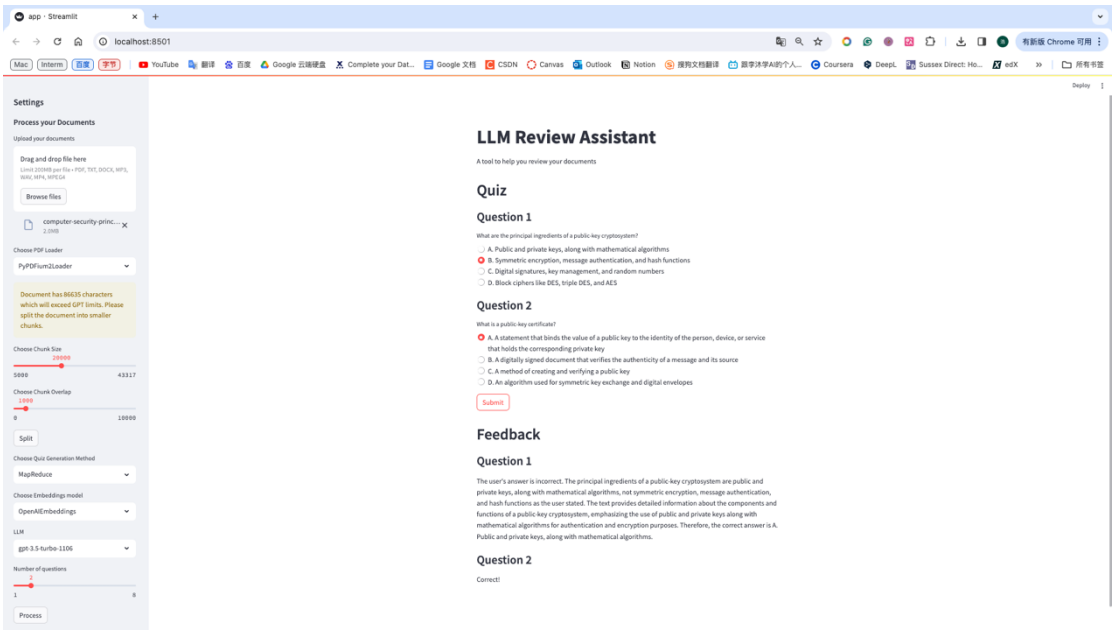



*Figure 4 | Front-End Page of this project.*

*Figure 4* shows the front page of this project. Details of different parts are shown below. The figure shows an example of the use of this application, including generated quizzes and feedback after the user uploads a file.

### 3.1.1. Setting Part

The Settings section is on the left side of the page. It consists of seven sections: File Upload, File Loader, File Splitter, Quiz Generation Method Selection, Embedded Model Selection, LLM Selection, and Number of Questions Selection, as shown in *Figure 5*.

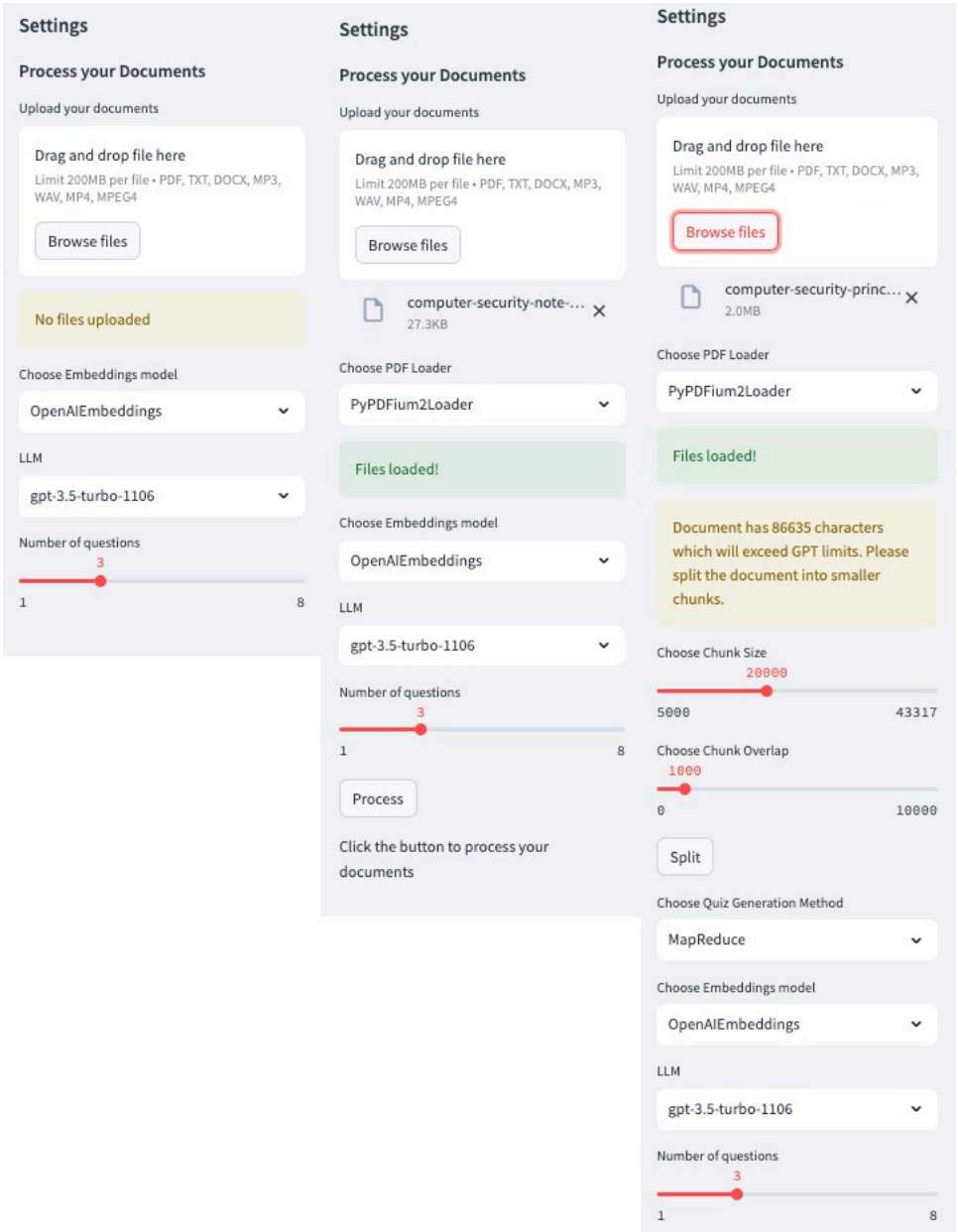



*Figure 5 | Setting part in two cases depending on the uploaded text length. The picture on the left is the default setting, the picture in the middle is the case when the length of the uploaded text is less than the LLM limit, and the picture is the case when the length exceeds the LLM limit, which needs to split the text and choose quiz generation method.*

### 3.1.1.1. File Uploader

The file upload part allows the user to upload one text, audio or video file from the user's computer, such as TXT, PDF, DOCX, MP3, WAV, MP4 or MPEG4 format files. As shown in line 5 of *Figure 6*, this application uses Streamlit's file_uploader to implement the file uploading interface and functionality.

```
1  with st.sidebar:
2      st.header("Settings")
3      st.subheader("Process your Documents")
4      # Upload files
5      uploaded_file = st.file_uploader("Upload your documents", accept_multiple_files=False, type=["pdf", "txt", "docx", "mp3", "wav", "mp4"])
6      # Select the loader
7      if uploaded_file is None:
8          st.warning("No files uploaded")
9      elif uploaded_file.name.endswith(".pdf"):
10         loader = st.selectbox("Choose PDF Loader", ["PyPDFium2Loader", "UnstructuredPDFLoader"])
11     elif uploaded_file.name.endswith(".txt"):
12         loader = st.selectbox("Choose Text Loader", ["TextLoader", "UnstructuredFileLoader"])
13     elif uploaded_file.name.endswith(".docx"):
14         loader = st.selectbox("Choose Word Document Loader", ["Docx2txtLoader", "UnstructuredWordDocumentLoader"])
15     elif uploaded_file.name.endswith(".mp3") or uploaded_file.name.endswith(".wav") or uploaded_file.name.endswith(".mp4"):
16         loader = "WhisperLoader"
17     else:
18         st.warning("Unsupported file type")
```

*Figure 6 | Code for file uploader and file loader selection.*

### 3.1.1.2. File Loader

After a user uploads a file, it must be converted to plain text first. This application uses Streamlit's selectbox to implement file loader selection, as shown in lines 7-18 of *Figure 6*. As shown in *Figure 7*, for TXT format files, you can choose TextLoader or UnstructuredFileLoader in the LangChain document loader component. For PDF format files, you can choose PyPDFium2Loader or UnstructuredPDFLoader in the LangChain document loader component. For DOCX format files, you can choose Docx2txtLoader or UnstructuredWordDocumentLoader in the LangChain document loader component. For the remaining audio and video files, there is no need to select a document loader. The application will automatically transcribe the audio using the Whisper model [5]. The difference between document loaders will be explained in detail in Section 3.2.2.

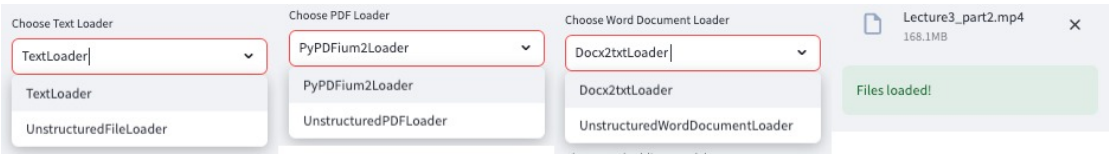



*Figure 7 | The first picture is the TXT format case, the second is the PDF format case, the third is the DOCX format case, and the last is the audio or video case.*

### 3.1.1.3. File Splitter

When the length of the text exceeds the limit of LLM, it will be split into chunks using RecursiveCharacterTextSplitter in the document splitter component of LangChain. As shown in the last picture in *Figure 5*, the user needs to select two parameters: chunk size and chunk overlap. This application enables parameter selection via Streamlit's slider, as shown in lines 5 and 6 of *Figure 8*.

```
1  # Split the files if it is too large
2  if st.session_state.large_file and st.session_state.doc:
3      st.warning(f"Document has {len(st.session_state.doc.page_content)} characters which will exceed GPT limits. Please split the document into smaller chunks.")
4      print(f"The total number of characters in {uploaded_file.name} is {len(st.session_state.doc.page_content)}")
5      chunk_size = st.slider("Choose Chunk Size", min_value=5000, max_value=min(len(st.session_state.doc.page_content)//2, 50000), value=20000, step=1000)
6      chunk_overlap = st.slider("Choose Chunk Overlap", min_value=0, max_value=chunk_size//2, value=1000, step=500)
7      if st.button("Split"):
8          st.spinner("Splitting files...")
9          st.session_state.chunks = chuk_files([st.session_state.doc], chunk_size, chunk_overlap)
10
11         st.success("Done!")
12     # Select the quiz generation method
13     st.session_state.generate_method = st.selectbox("Choose Quiz Generation Method", ["MapReduce", "Clustering", "Stuff"])
14
15 # Select the embeddings model
16 st.session_state.embeddings = st.selectbox("Choose Embeddings model", ["OpenAIEmbeddings", "HuggingFaceInstructEmbeddings"])
17 # Select the LLM model
18 st.session_state.llm_name = st.selectbox("LLM", ["gpt-3.5-turbo-1106", "gpt-4-1106-preview"])
19 # Select the number of questions
20 st.session_state.n_questions = st.slider("Number of questions", min_value=1, max_value=8, value=3, step=1)
```

*Figure 8 | Code for file splitter parameter selection, quiz generation methods selection, embedding model selection, LLM selection and number of question selection.*

### 3.1.1.4. Quiz Generation Method Selection

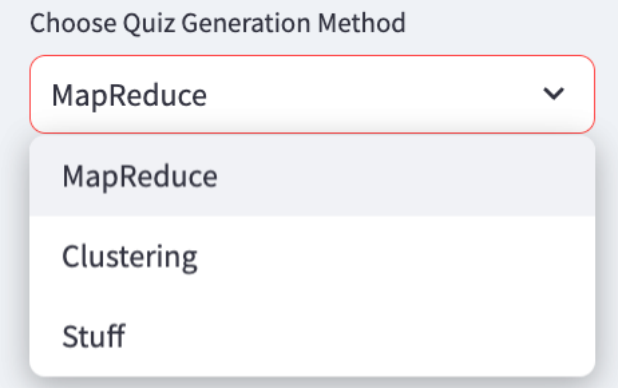



*Figure 9 | Three quiz generation methods when text length exceeds LLM limit.*

As shown in *Figure 5*, when the text length is within the LLM limit, this application will use Stuff to generate a quiz by default, so you do not need to choose the quiz generation method. When the text length exceeds the LLM limit, you must select the quiz generation method among the three methods of stuff, map reduce and clustering in *Figure 9*. The detailed differences between the three methods will be described in Section 3.2.6. This application implements the selection of the generation method through Streamlit's selectbox, as shown in line 13 of *Figure 8*.

### 3.1.1.5. Embedding Model Selection

To use LLM to generate feedback based on textual content via Retrieval Augmented Generation (RAG) [15], plain text must first be converted into word vectors in a high-dimensional vector space via an embedding model and then stored in the vector database Chroma. In addition, the quiz generation method of clustering also requires the use of word vectors. As shown in *Figure 10*, there are two embedding models, OpenAIEmbeddings [11] and HuggingFaceInstructEmbeddings [16]. The detailed differences between the embedding models will be described in Section 3.2.4. This application enables the selection of the embedded model through Streamlit's selectbox, as shown in line 16 of *Figure 8*.

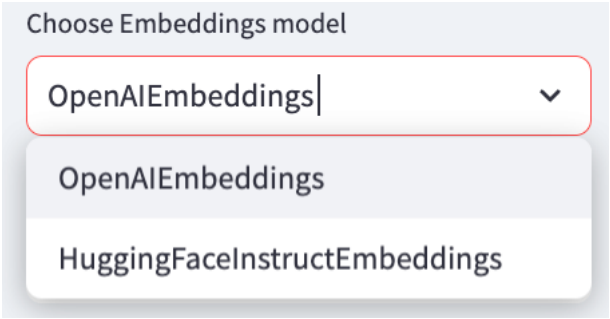



*Figure 10 | Embedding model selection.*

### 3.1.1.6. LLM and Number of Question Selection

This application provides two OpenAI GPT models: gpt-3.5-turbo-1106 and gpt-4-1106-preview [11], as shown in *Figure 11*. It implements the selection of the LLM through Streamlit's selectbox, as shown in line 18 of Figure 8, and the selection of question number through Streamlit's slider, as shown in line 20 of *Figure 8*.

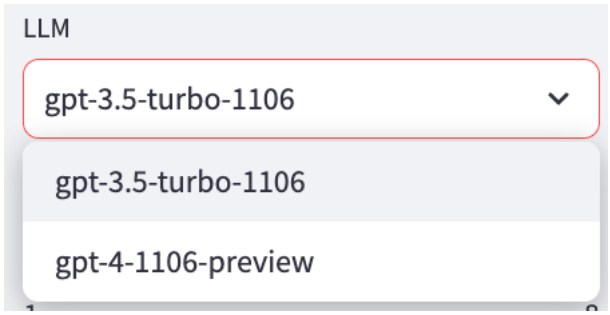



*Figure 11 | LLM Selection.*

### 3.1.2. Quiz Part

```
1  st.title("LLM Review Assistant")
2  st.write("A tool to help you review your documents")
3  st.header("Quiz")
4  if not st.session_state.quiz:
5      st.write("No quiz generated")
6  else:
7      # Display the quiz
8      for i, question in enumerate(st.session_state.quiz):
9          st.subheader(f"Question {i+1}")
10         st.session_state.user_answer[i] = st.radio(question["question"], question["options"], key=f"question_{i}", index=None)
11
12     if st.button("Submit"):
13         # Generate feedback for wrong answers
14         for i, question in enumerate(st.session_state.quiz):
15             if st.session_state.user_answer[i] == question["answer"]:
16                 st.session_state.feedback[i] = "Correct!"
17             else:
18                 # use LLM to generate feedback on wrong answers
19                 if st.session_state.user_answer_old.get(i) != st.session_state.user_answer[i]:
20                     if st.session_state.large_file:
21                         # use RAG to generate feedback for large files
22                         st.session_state.feedback[i] = generate_feedback_retrieval(st.session_state.llm, st.session_state.db, question, st.session_state.user_answer[i])
23                     else:
24                         st.session_state.feedback[i] = generate_feedback(st.session_state.llm, st.session_state.doc, question, st.session_state.user_answer[i])
25                 st.session_state.user_answer_old[i] = st.session_state.user_answer[i]
26
27     # Display the feedback
28     st.header("Feedback")
29     if not st.session_state.feedback:
30         st.write("Submit your answers to get feedback")
31     else:
32         for i, question in enumerate(st.session_state.quiz):
33             st.subheader(f"Question {i+1}")
34             st.write(st.session_state.feedback.get(i, "Submit your answer to get feedback"))
```

*Figure 12 | Code for quiz part.*

#### 3.1.2.1. Different Text Formats

This application uses Streamlit's title, header, subheader, text, success, warning, and spinner to achieve different text formats. For example, lines 2, 3, 8, and 18 in *Figure 6*, lines 3 and 8 in *Figure 8*, and lines 1-3, 5, 9, 28, 30, 33, and 34 in *Figure 12*. The spinner is used to display the state of an executing block of code, e.g., lines 8 and 9 in *Figure 8* will display the text in line 8 during the execution of line 9.

#### 3.1.2.2. Quiz

This application implements the radio widget via Streamlit's radio and stores the user-selected answer in a shared variable in the session state, as shown in line 10 of *Figure 12*.

#### 3.1.2.3. Button

This application implements the button widget through Streamlit's button. This widget returns True when the user clicks on the button and False otherwise, as shown in line 12 of *Figure 12*.

## 3.2. Back-End Development

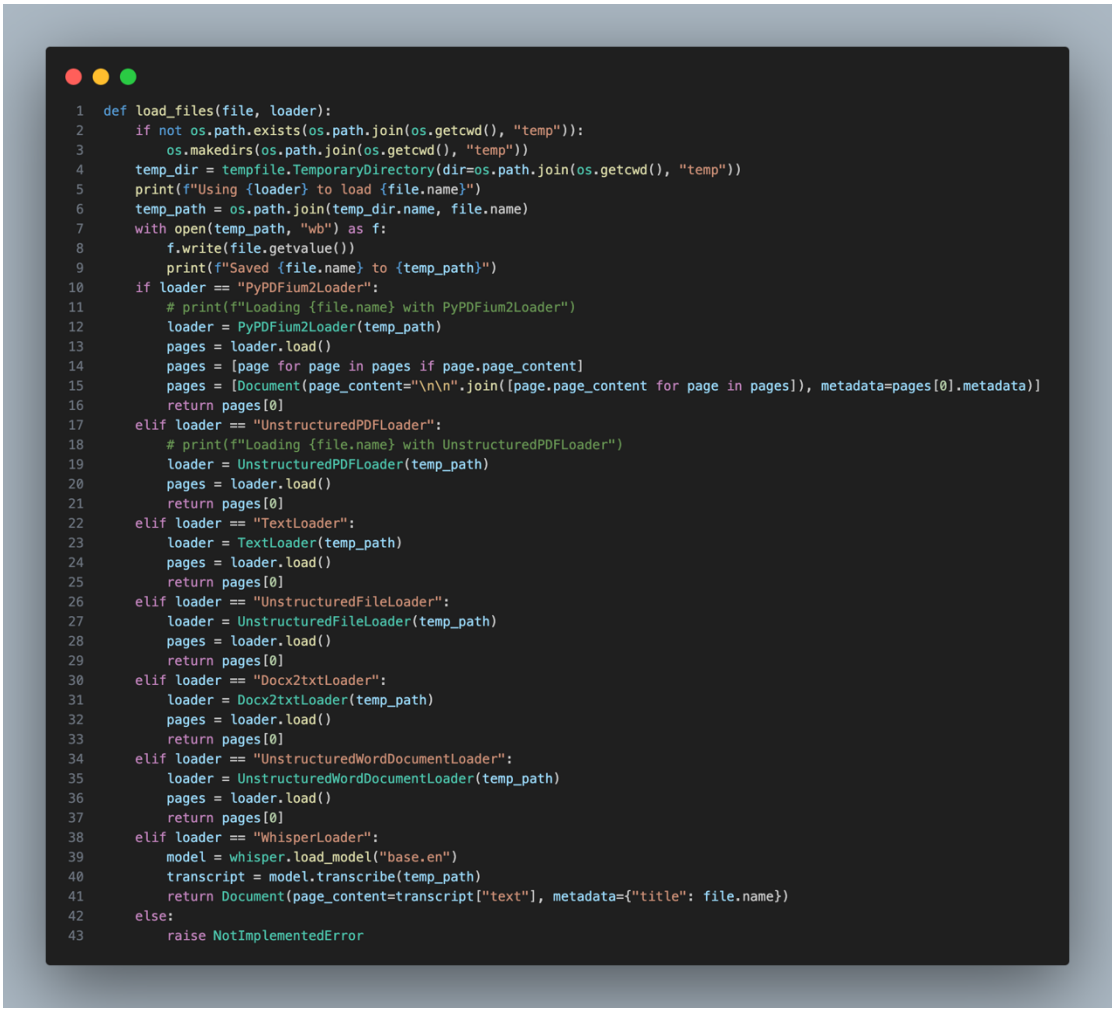
### 3.2.1. Environment and Session State Variables

As shown in line 2 of *Figure 13*, this line of code loads environment variables such as OPENAI_API_KEY set in the .env file through the dotenv library. Lines 4-34 in *Figure 13* create a series of variables in Streamlit's session state to store some persistent state.

```
1  def main():
2      load_dotenv()
3
4      #  Initialize parameters in session state
5      if 'filename' not in st.session_state:
6          st.session_state.filename = None
7      if 'doc' not in st.session_state:
8          st.session_state.doc = None
9      if 'chunks' not in st.session_state:
10         st.session_state.chunks = []
11     if 'embeddings' not in st.session_state:
12         st.session_state.embeddings = "OpenAIEmbeddings"
13     if 'llm_name' not in st.session_state:
14         st.session_state.llm_name = None
15     if 'n_questions' not in st.session_state:
16         st.session_state.n_questions = 3
17     if 'generate_method' not in st.session_state:
18         st.session_state.generate_method = "MapReduce"
19     if 'llm' not in st.session_state:
20         st.session_state.llm = None
21     if 'large_file' not in st.session_state:
22         st.session_state.large_file = False
23     if 'doc_limit' not in st.session_state:
24         st.session_state.doc_limit = 63_000
25     if 'db' not in st.session_state:
26         st.session_state.db = None
27     if 'quiz' not in st.session_state:
28         st.session_state.quiz = []
29     if 'user_answer' not in st.session_state:
30         st.session_state.user_answer = {}
31     if 'user_answer_old' not in st.session_state:
32         st.session_state.user_answer_old = {}
33     if 'feedback' not in st.session_state:
34         st.session_state.feedback = {}
```

*Figure 13 | Code for environment and session state variables.*

### 3.2.2. Load File and Judge Large File



```python
def load_files(file, loader):
    if not os.path.exists(os.path.join(os.getcwd(), "temp")):
        os.makedirs(os.path.join(os.getcwd(), "temp"))
    temp_dir = tempfile.TemporaryDirectory(dir=os.path.join(os.getcwd(), "temp"))
    print(f"Using {loader} to load {file.name}")
    temp_path = os.path.join(temp_dir.name, file.name)
    with open(temp_path, "wb") as f:
        f.write(file.getvalue())
        print(f"Saved {file.name} to {temp_path}")
    if loader == "PyPDFium2Loader":
        # print(f"Loading {file.name} with PyPDFium2Loader")
        loader = PyPDFium2Loader(temp_path)
        pages = loader.load()
        pages = [page for page in pages if page.page_content]
        pages = [Document(page_content="\n\n".join([page.page_content for page in pages]), metadata=pages[0].metadata)]
        return pages[0]
    elif loader == "UnstructuredPDFLoader":
        # print(f"Loading {file.name} with UnstructuredPDFLoader")
        loader = UnstructuredPDFLoader(temp_path)
        pages = loader.load()
        return pages[0]
    elif loader == "TextLoader":
        loader = TextLoader(temp_path)
        pages = loader.load()
        return pages[0]
    elif loader == "UnstructuredFileLoader":
        loader = UnstructuredFileLoader(temp_path)
        pages = loader.load()
        return pages[0]
    elif loader == "Docx2txtLoader":
        loader = Docx2txtLoader(temp_path)
        pages = loader.load()
        return pages[0]
    elif loader == "UnstructuredWordDocumentLoader":
        loader = UnstructuredWordDocumentLoader(temp_path)
        pages = loader.load()
        return pages[0]
    elif loader == "WhisperLoader":
        model = whisper.load_model("base.en")
        transcript = model.transcribe(temp_path)
        return Document(page_content=transcript["text"], metadata={"title": file.name})
    else:
        raise NotImplementedError
```

*Figure 14 | Code for loading files function.*

*Figure 14* shows a function that loads the text into plain text using a user-selected file loader. Line 6 of *Figure 15* calls the function in *Figure 14* and stores the plain text it returns in a variable in the session state. Lines 15 and 16 of *Figure 15* will determine if the text length exceeds the LLM limit.

*Figure 7* shows another unstructured file loader available in addition to the normal file loader for TXT, PDF, and DOCX format files. The difference between the two types of file loaders is the speed and quality of reading the file. *Figure 16* and *Figure 17* show that an unstructured file loader loads files slower but gets higher read quality.

For audio and video files, the audio in the file is transcribed into plain text using the Whisper model [5], which comes in five sizes: tiny, base, small, medium, and large. For a test video, the base model takes 1m 42.2s and the large model takes 28m 12.2s. *Figure 18* shows the transcription results of the two size models. It is clear that the large size model is more accurate than the base model, but the large model can occasionally be too sensitive.

```python
# Load the files
if uploaded_file and uploaded_file.name != st.session_state.filename:
    with st.spinner("Loading files..."):
        st.session_state.filename = uploaded_file.name
        print(f"Uploaded files: {uploaded_file.name}")
        st.session_state.doc = load_files(uploaded_file, loader)
        print(f"Loaded {uploaded_file.name} with {loader}")

        # Show if the files are loaded
        if st.session_state.doc:
            st.success("Files loaded!")
        else:
            st.warning("No files uploaded")
    # Check if the file is too large
    if len(st.session_state.doc.page_content) > 63_000:
        st.session_state.large_file = True
```

*Figure 15 | Code for loading files and judging large files in the main function.*

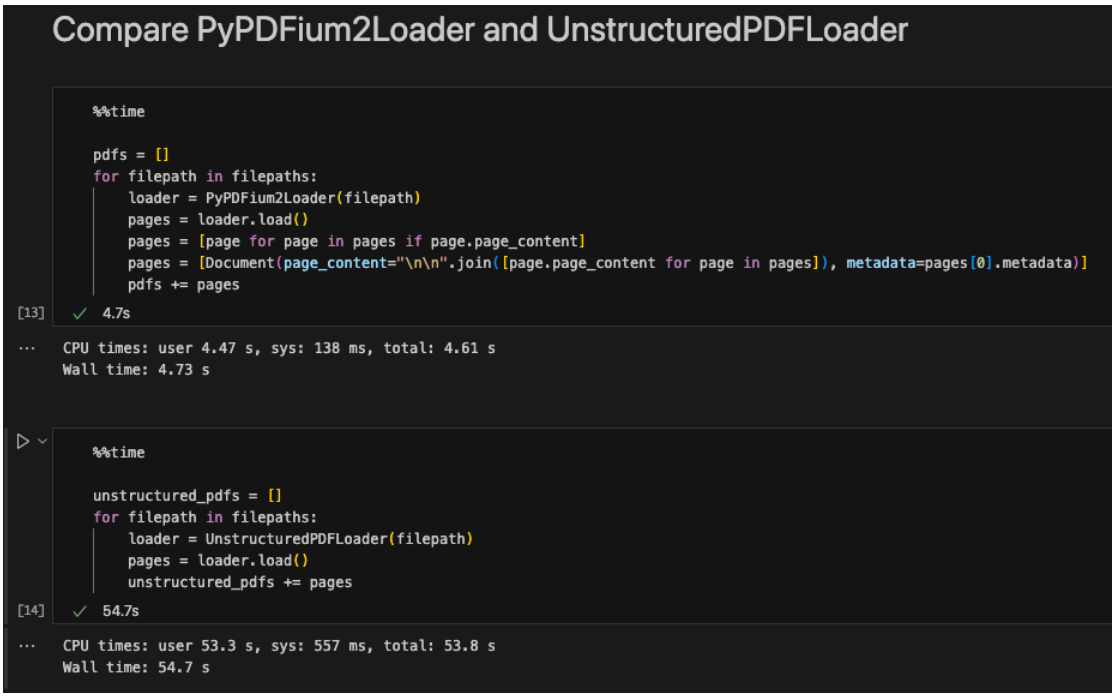



*Figure 16 | Compare the time cost of PyPDFium2Loader and UnstructuredPDFLoader.*

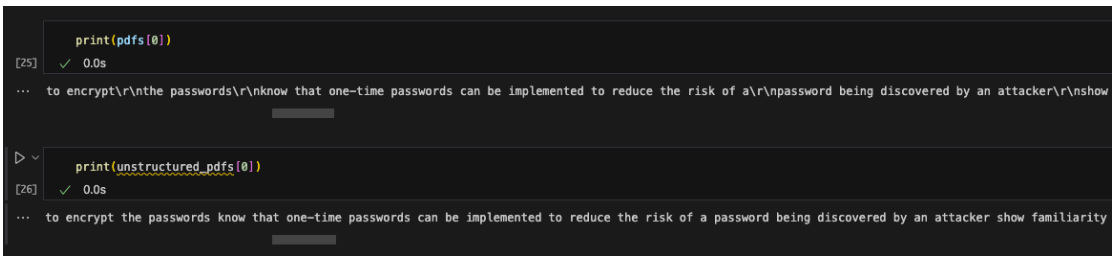



*Figure 17 | Output of PyPDFium2Loader and UnstructuredPDFLoader.*

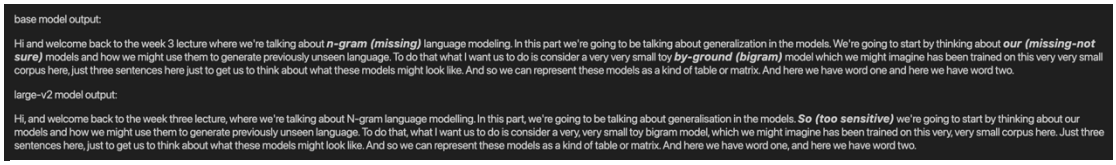



*Figure 18 | Compare the transcription of base and large whisper model.*

## 3.2.3. Chunk Files

As shown in *Figure 8*, when the text length exceeds the LLM limit, the user is asked to select chunk size and chunk overlap. The chunk size ranges from 5000 to half of the text length, and the chunk overlap ranges from 0 to half of the chunk size. *Figure 19* shows the function that uses the user-selected chunk size and chunk overlap to chunk files.

```python
def chuk_files(doc, chunk_size, chunk_overlap):
    print(f"Using RecursiveCharacterTextSplitter with chunk_size={chunk_size} and chunk_overlap={chunk_overlap}")
    splitter = RecursiveCharacterTextSplitter(separators=["\n", "\r\n", "\r", "\t", "\n\n"], chunk_size=chunk_size, chunk_overlap=chunk_overlap)
    chunks = splitter.split_documents(doc)
    print(f"Split {len(doc)} pages into {len(chunks)} chunks")
    return chunks
```

*Figure 19 | Code for chunking files function.*

## 3.2.4. Embedding

```python
# Set the document limit
if st.session_state.llm_name == "gpt-4-1106-preview":
    st.session_state.doc_limit = 511_000
elif st.session_state.llm_name == "gpt-3.5-turbo-1106":
    st.session_state.doc_limit = 63_000

if (st.session_state.large_file and st.session_state.chunks) or (not st.session_state.large_file and st.session_state.doc):
    if st.button("Process"):
        if uploaded_file:
            # Embed the chunks/documents
            with st.spinner("Embedding ..."):
                if st.session_state.large_file:
                    st.session_state.db = embed(st.session_state.chunks, st.session_state.embeddings)
                else:
                    st.session_state.db = embed([st.session_state.doc], st.session_state.embeddings)
            # Load the LLM
            with st.spinner("Loading LLM..."):
                print(f"Loading {st.session_state.llm_name}")
                st.session_state.llm = load_llm(st.session_state.llm_name)
            # Generate quiz based on the method selected
            with st.spinner("Generating quiz..."):
                print(f"Generating quiz using {st.session_state.llm_name}")
                if st.session_state.large_file:
                    if st.session_state.generate_method == "Stuff":
                        max_chunks = st.session_state.doc_limit // chunk_size
                        num_chunks = len(st.session_state.chunks)
                        if st.session_state.n_questions > (num_chunks // max_chunks):
                            for i in range(num_chunks // max_chunks):
                                st.session_state.quiz += generate_quiz(st.session_state.llm, st.session_state.chunks[i*max_chunks:(i+1)*max_chunks], st.session_state.n_questions // (num_chunks//max_chunks))
                            if len(st.session_state.quiz) < st.session_state.n_questions:
                                st.session_state.quiz += generate_quiz(st.session_state.llm, st.session_state.chunks[(num_chunks//max_chunks)*max_chunks:], st.session_state.n_questions - len(st.session_state.quiz))
                        else:
                            st.session_state.quiz = generate_quiz(st.session_state.llm, st.session_state.chunks[len(st.session_state.chunks)//2:len(st.session_state.chunks)//2+max_chunks], st.session_state.n_questions)
                    elif st.session_state.generate_method == "MapReduce":
                        st.session_state.quiz = generate_quiz_map_reduce(st.session_state.llm, st.session_state.chunks, st.session_state.n_questions)
                    elif st.session_state.generate_method == "Clustering":
                        # alter the number of clusters to the number of questions
                        if st.session_state.n_questions*chunk_size >  st.session_state.doc_limit:
                            max_chunks = st.session_state.doc_limit // chunk_size
                            for i in range(st.session_state.n_questions // max_chunks):
                                st.session_state.quiz += generate_quiz_clustering(st.session_state.llm, st.session_state.db, max_chunks)
                            st.session_state.quiz += generate_quiz_clustering(st.session_state.llm, st.session_state.db, st.session_state.n_questions % max_chunks)
                        else:
                            st.session_state.quiz = generate_quiz_clustering(st.session_state.llm, st.session_state.db, st.session_state.n_questions)

                    else:
                        raise NotImplementedError
                else:
                    st.session_state.quiz = generate_quiz(st.session_state.llm, [st.session_state.doc], st.session_state.n_questions)

            st.success("Done!")
            print(f"Generated quiz: {st.session_state.quiz}")
        else:
            st.write("No files uploaded")
    else:
        st.write("Click the button to process your documents")
```

*Figure 20 | Code for embedding, loading LLM and generating LLM.*

```python
def embed(docs, embeddings):
    print(f"Using {embeddings}")
    if embeddings == "HuggingFaceInstructEmbeddings":
        embeddings = HuggingFaceInstructEmbeddings(model_name="hkunlp/instructor-xl")
    elif embeddings == "OpenAIEmbeddings":
        embeddings = OpenAIEmbeddings()
    else:
        raise NotImplementedError
    # vectors = embeddings.embed_documents([chunk.page_content for chunk in chunks])
    db = Chroma.from_documents(docs, embeddings)
    return db
```

*Figure 21 | Code for embedding function.*

*Figure 21* shows the function that converts plain text into word vectors via a user-selected embedding model and stores them in the vector database Chroma. There are two embedding models to choose from. HuggingFaceInstructEmbeddings [16] runs locally on the user's machine, which is a little slower but has no extra expense. OpenAIEmbeddings calls OpenAI's Embedding API[11], which is fast but charges a fee. Lines 10-15 in *Figure 20* call the function in *Figure 21*.

### 3.2.5. Load LLM

```python
def load_llm(llm_name):
    if llm_name == "gpt-3.5-turbo-1106":
        llm = ChatOpenAI(model_name="gpt-3.5-turbo-1106")
    elif llm_name == "gpt-4-1106-preview":
        llm = ChatOpenAI(model_name="gpt-4-1106-preview")
    else:
        raise NotImplementedError
    return llm
```

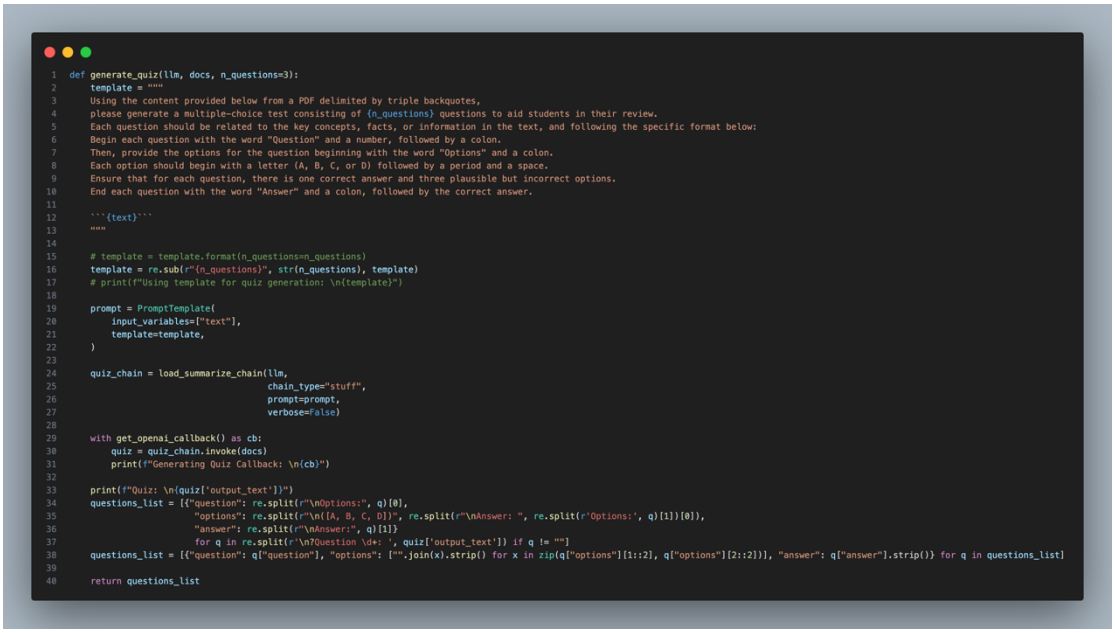
*Figure 22 | Code for loading LLM.*

*Figure 22* shows the function that loads LLM. The gpt-3.5-turbo-1106 and gpt-4-1106-preview have a context window limit of 16,385 and 128,000 tokens, respectively.[7] But the gpt-4-1106-preview API is 20 times more expensive than the gpt-3.5-turbo-1106 API. 1106. Since gpt-4-1106-preview API is too expensive, I default to using gpt-3.5-turbo-1106. Usually, a token has about 4-5 characters, and the template length is about 1,000 characters, so the context length needs to be less than $16{,}385 * 4 - 1{,}000 = 64{,}540$ tokens when choosing gpt-3.5-turbo-1106. So, I set the text length limit to 63,000.

## 3.2.6. Generate Quiz

As shown in lines 20-52 of *Figure 20*, when the text length is within the LLM limit, this application will use the stuff method to generate a quiz by default. When the length exceeds the LLM limit, the corresponding function will be called to generate a quiz according to the user-selected generation method.

### 3.2.6.1. Stuff



```python
def generate_quiz(llm, docs, n_questions=3):
    template = """
    Using the content provided below from a PDF delimited by triple backquotes,
    please generate a multiple-choice test consisting of {n_questions} questions to aid students in their review.
    Each question should be related to the key concepts, facts, or information in the text, and following the specific format below:
    Begin each question with the word "Question" and a number, followed by a colon.
    Then, provide the options for the question beginning with the word "Options" and a colon.
    Each option should begin with a letter (A, B, C, or D) followed by a period and a space.
    Ensure that for each question, there is one correct answer and three plausible but incorrect options.
    End each question with the word "Answer" and a colon, followed by the correct answer.

    ```{text}```
    """

    # template = template.format(n_questions=n_questions)
    template = re.sub(r"{n_questions}", str(n_questions), template)
    # print(f"Using template for quiz generation: \n{template}")

    prompt = PromptTemplate(
        input_variables=["text"],
        template=template,
    )

    quiz_chain = load_summarize_chain(llm,
                                      chain_type="stuff",
                                      prompt=prompt,
                                      verbose=False)

    with get_openai_callback() as cb:
        quiz = quiz_chain.invoke(docs)
        print(f"Generating Quiz Callback: \n{cb}")

    print(f"Quiz: \n{quiz['output_text']}")
    questions_list = [{"question": re.split(r"\nOptions:", q)[0],
                       "options": re.split(r"\n([A, B, C, D])", re.split(r"\nAnswer: ", re.split(r'Options:', q)[1])[0]),
                       "answer": re.split(r"\nAnswer:", q)[1]}
                      for q in re.split(r'\n?Question \d+: ', quiz['output_text']) if q != ""]
    questions_list = [{"question": q["question"], "options": ["".join(x).strip() for x in zip(q["options"][1::2], q["options"][2::2])], "answer": q["answer"].strip()} for q in questions_list]

    return questions_list
```

*Figure 23 | Stuff Quiz Generation Function.*

[7] https://platform.openai.com/docs/models
```

The stuff generation method inserts the context text into the prompt template and passes it directly to the LLM. As shown in Lines 24-33 of *Figure 20*, the maximum chunk number that can be accepted by a single LLM request is $LLM\ limit \mid chunk\ size$, so the question number that can be generated by a single request is $question\ number \mid (chunk\ number \mid maximum\ chunk\ number\ per\ request)$. "|" in the equation denotes integer division. When the question number set by the user is greater than the chunk number used by a single LLM request, i.e. $chunk\ number \mid maximum\ chunk\ number\ per\ request$, then multiple LLM requests are required, and each request generates $question\ number \mid (chunk\ number \mid maximum\ chunk\ number\ per\ request)$ questions, and the remaining questions are generated in the last request. When the question number set by the user is less than the chunk number used by a single LLM request, then the number of questions generated by one request will be less than 1. So, the maximum number of chunks in the middle of the context text will be taken to generate a quiz at once.

*Figure 23* shows the stuff quiz generation function. Lines 2-22 define the prompt template used to generate the quiz and insert the context into the prompt template; lines 24-27 generate a new quiz generation chain through the summary chain of the LangChain with the custom prompt template; lines 29-31 pass the above prompt template to the OpenAI API to get the response from the LLM through quiz generation chain, lines 33-38 process the LLM's response through regularisation and store quiz into a list in the order of question, options, answer for each MCQ.

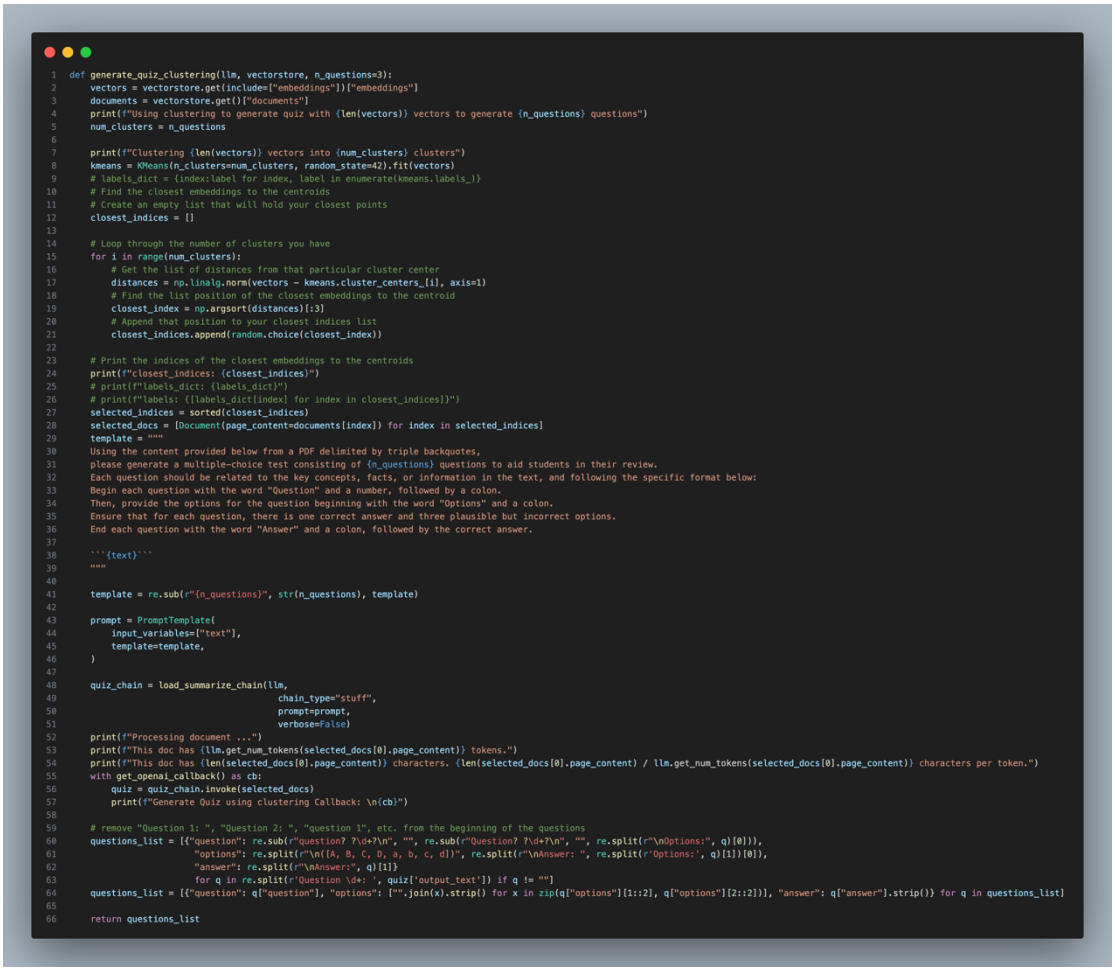
### 3.2.6.2. Map Reduce

```
1  def generate_quiz_map_reduce(llm, docs, n_questions=3):
2      map_prompt = """
3      Summarize the following text delimited by triple backquotes.
4      The summary should contain the key points of the text and help students revise.
5      ```{text}```
6      """
7
8      map_prompt_template = PromptTemplate(template=map_prompt, input_variables=["text"])
9
10     combine_prompt = """
11     Using the summary provided below from a PDF delimited by triple backquotes,
12     please generate a multiple-choice test consisting of {n_questions} questions to aid students in their review.
13     Each question should be related to the key concepts, facts, or information in the text, and following the specific format below:
14     Begin each question with the word "Question" and a number, followed by a colon.
15     Then, provide the options for the question beginning with the word "Options" and a colon.
16     Each option should begin with a letter (A, B, C, or D) followed by a period and a space.
17     Ensure that for each question, there is one correct answer and three plausible but incorrect options.
18     End each question with the word "Answer" and a colon, followed by the correct answer.
19
20     ```{text}```
21     """
22     combine_prompt = re.sub(r"{n_questions}", str(n_questions), combine_prompt)
23     combine_prompt_template = PromptTemplate(template=combine_prompt, input_variables=["text"])
24
25     quiz_chain = load_summarize_chain(llm,
26                                       chain_type="map_reduce",
27                                       map_prompt=map_prompt_template,
28                                       combine_prompt=combine_prompt_template,
29                                       verbose=False)
30
31     with get_openai_callback() as cb:
32         quiz = quiz_chain.invoke(docs)
33         print(f"Generate Quiz using map reduce Callback: \n{cb}")
34
35     # remove "Question 1: ", "Question 2: ", "question 1", etc. from the beginning of the questions
36     questions_list = [{"question": re.sub(r"question? ?\d+?\n", "", re.sub(r"Question? ?\d+?\n", "", re.split(r"\nOptions:", q)[0])),
37                        "options": re.split(r"\n([A, B, C, D])", re.split(r"\nAnswer: ", re.split(r'Options:', q)[1])[0]),
38                        "answer": re.split(r"\nAnswer:", q)[1]}
39                       for q in re.split(r'Question \d+: ', quiz['output_text']) if q != ""]
40     questions_list = [{"question": q["question"], "options": ["".join(x).strip() for x in zip(q["options"][1::2], q["options"][2::2])], "answer": q["answer"].strip()} for q in questions_list]
```

*Figure 24 | Map Reduce Quiz Generation Function.*

The map-reduce method first summarises the context text to reduce its length and then generates a quiz using the summary output as the context. As shown in lines 34 and 35 in *Figure 20*, this application calls the map-reduce quiz generation function to generate a quiz when the user selects the map-reduce method of quiz generation. *Figure 24* shows the map-reduce quiz generation function. Lines 2-8 define the summary prompt template, lines 10-23 define the quiz generation prompt template, lines 25-29 use the map-reduce type of LangChain's summary chain with the above two prompt templates to generate a new map-reduce quiz generation chain, lines 31-33 pass the above prompt templates to the OpenAI API to get the responses from LLM through the map-reduce chain. Lines 25-40 process the responses from LLM through regularisation and store the responses into a quiz list in the order of questions, options and answers for each MCQ.

### 3.2.6.3. Clustering



```
def generate_quiz_clustering(llm, vectorstore, n_questions=3):
    vectors = vectorstore.get(include=["embeddings"])["embeddings"]
    documents = vectorstore.get()["documents"]
    print(f"Using clustering to generate quiz with {len(vectors)} vectors to generate {n_questions} questions")
    num_clusters = n_questions

    print(f"Clustering {len(vectors)} vectors into {num_clusters} clusters")
    kmeans = KMeans(n_clusters=num_clusters, random_state=42).fit(vectors)
    # labels_dict = {index:label for index, label in enumerate(kmeans.labels_)}
    # Find the closest embeddings to the centroids
    # Create an empty list that will hold your closest points
    closest_indices = []

    # Loop through the number of clusters you have
    for i in range(num_clusters):
        # Get the list of distances from that particular cluster center
        distances = np.linalg.norm(vectors - kmeans.cluster_centers_[i], axis=1)
        # Find the list position of the closest embeddings to the centroid
        closest_index = np.argsort(distances)[:3]
        # Append that position to your closest indices list
        closest_indices.append(random.choice(closest_index))

    # Print the indices of the closest embeddings to the centroids
    print(f"closest_indices: {closest_indices}")
    # print(f"labels_dict: {labels_dict}")
    # print(f"labels: {[labels_dict[index] for index in closest_indices]}")
    selected_indices = sorted(closest_indices)
    selected_docs = [Document(page_content=documents[index]) for index in selected_indices]
    template = """
    Using the content provided below from a PDF delimited by triple backquotes,
    please generate a multiple-choice test consisting of {n_questions} questions to aid students in their review.
    Each question should be related to the key concepts, facts, or information in the text, and following the specific format below:
    Begin each question with the word "Question" and a number, followed by a colon.
    Then, provide the options for the question beginning with the word "Options" and a colon.
    Ensure that for each question, there is one correct answer and three plausible but incorrect options.
    End each question with the word "Answer" and a colon, followed by the correct answer.

    ```{text}```
    """

    template = re.sub(r"{n_questions}", str(n_questions), template)

    prompt = PromptTemplate(
        input_variables=["text"],
        template=template,
    )

    quiz_chain = load_summarize_chain(llm,
                                      chain_type="stuff",
                                      prompt=prompt,
                                      verbose=False)
    print(f"Processing document ...")
    print(f"This doc has {llm.get_num_tokens(selected_docs[0].page_content)} tokens.")
    print(f"This doc has {len(selected_docs[0].page_content)} characters. {len(selected_docs[0].page_content) / llm.get_num_tokens(selected_docs[0].page_content)} characters per token.")
    with get_openai_callback() as cb:
        quiz = quiz_chain.invoke(selected_docs)
        print(f"Generate Quiz using clustering Callback: \n{cb}")

    # remove "Question 1: ", "Question 2: ", "question 1", etc. from the beginning of the questions
    questions_list = [{"question": re.sub(r"question? ?\d+?\n", "", re.sub(r"Question? ?\d+?\n", "", re.split(r"\nOptions:", q)[0])),
                       "options": re.split(r"\n([A, B, C, D, a, b, c, d])", re.split(r"\nAnswer: ", re.split(r'Options:', q)[1])[0]),
                       "answer": re.split(r"\nAnswer:", q)[1]}
                      for q in re.split(r"Question \d+: ", quiz['output_text']) if q != ""]
    questions_list = [{"question": q["question"], "options": ["".join(x).strip() for x in zip(q["options"][1::2], q["options"][2::2])], "answer": q["answer"].strip()} for q in questions_list]

    return questions_list
```

*Figure 25 | Clustering Quiz Generation Function.*
```

The clustering test generation method clusters the word vectors of the split context chunks and selects the chunk whose word vector is closest to the centroid of each cluster as the context to generate the quiz. As shown in *Figure 20*, lines 36-44, the maximum chunk number that can be accepted in a single LLM request is $LLM\ limit \mid chunk\ size$, while one chunk corresponds to one question by default. Therefore, the clustering quiz generation function can be called directly when the generated question number the user selects is less than the maximum number of chunks acceptable for a single LLM request. When the generated question number is greater than the maximum chunk number that can be accepted in a single LLM request, it is necessary to make multiple LLM requests and generate $LLM\ limit \mid chunk\ size$ questions for each request until the last request generates the remaining questions that are less than the $LLM\ limit \mid chunk\ size$.

*Figure 25* shows the clustering quiz generation function. Lines 2 and 3 get the word vectors and corresponding text of each chunk from the vector database, line 8 clusters the word vectors of chunks by the KMeans algorithm, lines 10-21 randomly select an index randomly from three indexes of word vectors closest to the cluster centroid for each cluster, and lines 27 and 28 get the corresponding text chunks based on the indexes selected above. Lines 29-46 define the prompt template used to generate the quiz and insert the context into the prompt template, lines 48-51 generate a new quiz generation chain with the custom prompt template through the summary chain of the LangChain, lines 52-57 pass the above prompt template to the OpenAI API through the quiz generation chain to get the response from LLM, and finally Lines 59-64 process the LLM responses through regularisation and store the responses into a quiz list in the order of question, options and answer for each MCQ.

*Figure 26* shows a visualisation of the clustering results. This experiment split the test book into 142 text chunks, each corresponding to a 1536-dimensional word vector. By using the KMeans algorithm, these 1536-dimensional word vectors were effectively clustered into three categories. For visualisation, the 1536-dimensional word vectors first need to be downscaled to 2 dimensions, which is done by employing the TSNE algorithm. Subsequently, these dimensionality-reduced word vectors are visualised into a 2D coordinate system, and different clustering results are marked with different colours. It is clear from the figure that the KMeans algorithm clearly classifies the 142 text chunks into 3 different categories. This clustering method can efficiently select representative and differentiated text chunks for quiz generation, greatly significantly time and cost.

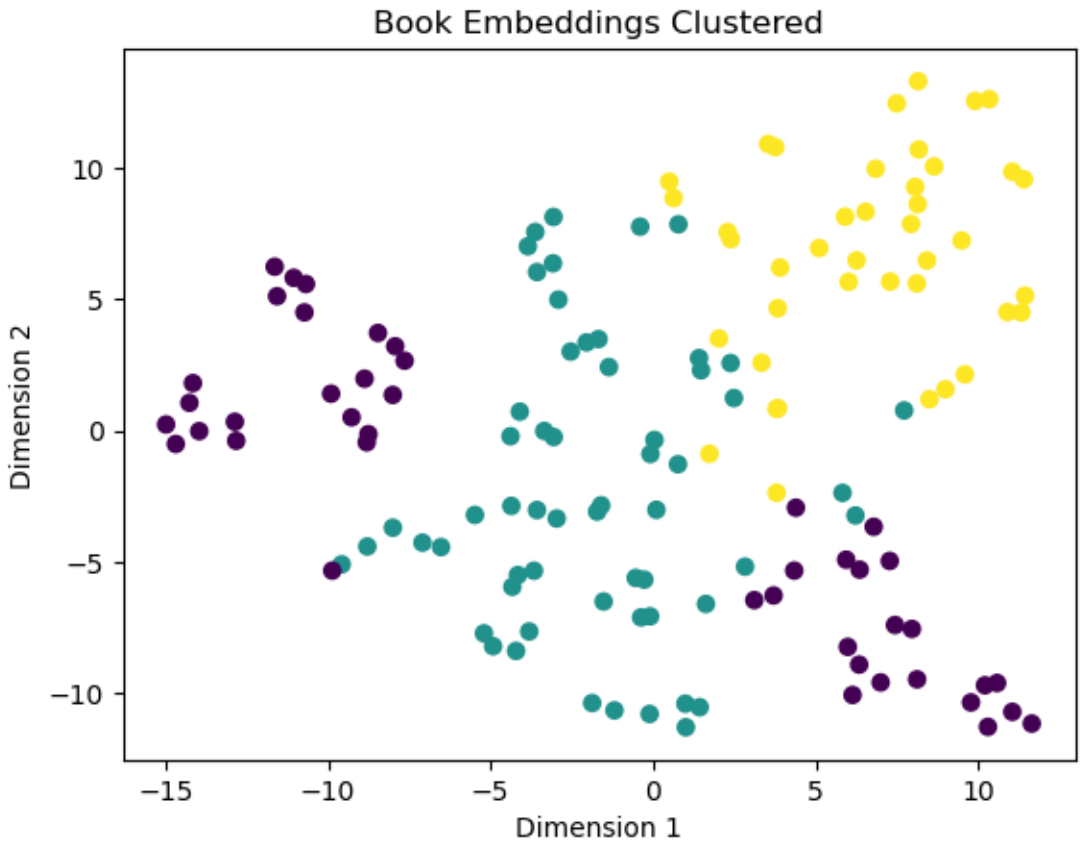



*Figure 26 | Visualisation of distribution of each cluster.*

#### 3.2.6.4. Discussion on Three Quiz Generation Methods

The context lengths of these three methods are different: the Map Reduce method is the length of the whole text plus the summary, the Stuff method is the length of the whole text, and the Clustering method is the question number multiplied by the chunk size. Therefore, the order of comparison of these three methods in terms of context length is Map Reduce > Stuff > Clustering. Accordingly, their time consumption and cost show the same order: Map Reduce > Stuff > Clustering.

Regarding the quality of the generated quiz, the Clustering method can pick representative chunks as contexts to generate the quiz. While the Map Reduce method may lose some details in the preliminary summarisation stage, the Stuff method slices the text by a fixed step size or randomly selects the intermediate text, which is more random and makes it harder to grasp the text's main point. Therefore, the order of comparison of these three methods in generating test quality is Clustering > Map Reduce > Stuff.

### 3.2.7. Generate Feedback

As shown in lines 12 to 25 in *Figure 12*, the application compares the answer the user selects with the correct answer after the user clicks the submit button. If the user answers correctly, the feedback message is "Correct! If the user's answer is incorrect and a new option is selected that differs from the previous one, the system calls the feedback generation function. The function generates feedback based on the content of the question, the correct answer, the user's choice, and the uploaded text. Specifically, if the length of the context does not exceed the limits of the LLM, the system will generate the feedback directly using the Stuff method; if it exceeds the limits, the feedback will be generated using the RAG method.

#### 3.2.7.1. Stuff

```python
def generate_feedback(llm, doc, question, user_answer):
    template = """
    Using the content provided below from a PDF delimited by triple backquotes,
    please provide feedback on the user's answer to the question below.

    Question: {question}
    Options: {options}
    Correct answer: {answer}
    User's answer: {user_answer}

    The feedback should be related to the key concepts, facts, or information in the text.

    ```{text}```

    Feedback:
    """

    template = re.sub(r"{question}", question["question"], template)
    template = re.sub(r"{options}", "; ".join(question["options"]), template)
    template = re.sub(r"{answer}", question["answer"], template)
    template = re.sub(r"{user_answer}", user_answer, template)
    # print(f"Using template for feedback: \n{template}")

    prompt = PromptTemplate(
        input_variables=["text"],
        template=template,
    )

    feedback_chain = load_summarize_chain(llm,
                                          chain_type="stuff",
                                          prompt=prompt,
                                          verbose=False)
    with get_openai_callback() as cb:
        feedback = feedback_chain.invoke([doc])
        print(f"Generating Feedback Callback: \n{cb}")

    # delete "Feedback:", "Feedback: ", "Feedback: \n" etc. from the beginning of the feedback
    feedback = re.sub(r"Feedback:? ?\n?", "", feedback['output_text'])
    print(f"Feedback: \n{feedback}")
    return feedback
```

*Figure 27 | Stuff Feedback Generation Function.*
```

*Figure 27* shows the Stuff feedback generation function. It is similar to the stuff quiz generator function in Section 3.2.6.1. The only difference is the definition of the prompt template in lines 2-27, where the question content, the correct answer and the user's choice are simultaneously written into the prompt template to create a new prompt template.

### 3.2.7.2. RAG

```python
def format_docs(docs):
    return "\n\n".join(doc.page_content for doc in docs)


def generate_feedback_retrieval(llm, vectorstore, question, user_answer):
    print(f"Generating feedback using retrieval for question: {question['question']}, user_answer: {user_answer}")


    retriever = vectorstore.as_retriever(search_type="similarity", search_kwargs={"k": 1})
    template = """
    Using the content provided below from a PDF delimited by triple backquotes,
    please provide feedback on the user's answer to the question below.

    Question: {question}
    Options: {options}
    Correct answer: {answer}
    User's answer: {user_answer}

    The feedback should be related to the key concepts, facts, or information in the text.

    ```{context}```

    Feedback:
    """
    # template = re.sub(r"{question}", question["question"], template)
    template = re.sub(r"{options}", "; ".join(question["options"]), template)
    template = re.sub(r"{answer}", question["answer"], template)
    template = re.sub(r"{user_answer}", user_answer, template)
    # print(f"Using template for feedback: \n{template}")
    custom_rag_prompt = PromptTemplate.from_template(template)

    rag_chain = (
        {"context": retriever | format_docs, "question": RunnablePassthrough()}
        | custom_rag_prompt
        | llm
        | StrOutputParser()
    )

    with get_openai_callback() as cb:
        feedback = rag_chain.invoke(question["question"])
        print(f"Generating Feedback using retrieval Callback: \n{cb}")

    # delete "Feedback:", "Feedback: ", "Feedback: \n" etc. from the beginning of the feedback
    feedback = re.sub(r"Feedback:? ?\n?", "", feedback)
    print(f"Feedback: \n{feedback}")
    return feedback
```

*Figure 28 | RAG Feedback Generation Function.*
```

The RAG feedback generation method utilises a vector database which stores the content of text blocks with word vectors as the basis for retrieval so that the most matching text blocks are retrieved based on the similarity of the question to generate feedback. As shown in *Figure 28*, line 9 of the code transforms the vector database into a retriever that is able to select the text chunk that best matches the input content based on similarity. Lines 10 to 30 define the prompt template used to generate the feedback and insert the options, correct answer and user answer into it. Lines 32 to 37 define the RAG chain, which includes the retriever, the customised prompt template, the LLM and the output parser. In lines 39 to 42 of the code, the question content is fed into the RAG chain defined above. Specifically, the retriever first finds the text chunk most similar to the question. Subsequently, the most similar text chunk is input into the prompt template along with the question to generate the complete prompt text. Afterwards, the prompt text is fed into the LLM to obtain a response. Finally, the LLM response is processed through the output parser. At line 44, the output of the custom RAG chain is cleaned up by regularisation to get the final feedback.

# 4. Future Work

Although students can already use this application to consolidate their knowledge, there is still much work to be done to improve this application.

## 4.1. Summary Uploaded Document

This application is currently capable of generating MCQs to assist users in consolidating their knowledge. If the uploaded files could be summarised into notes, this would further improve the efficiency of users' revision. However, after several attempts, it was found that due to the diverse detail levels of the files uploaded by users, the summarisation results usually fell short of expectations. Especially for class notes type files, key information is easily lost during the summarization process. For books or lecture audio files, the summarisation is often too detailed and the extraction of key information is unsatisfactory. Therefore, there is still a lot of potential for development in this application in terms of the ability to summarise uploaded files into notes, and future work could focus on improving the accuracy and relevance of the summaries. This will not only enhance the learning experience but also make the application more useful and effective.

## 4.2. Chat on the Uploaded Document

This application currently helps students to consolidate their knowledge mainly through quizzes, but sometimes the areas where the user is confused are not always covered by the test or the feedback given by the application is not enough to help the user understand the knowledge. So, if a discussion window based on uploaded text is added, it can better help the user to learn.

## 4.3. Other Quiz Format

Currently, the application only provides quizzes in the form of MCQs. This single form of quiz does not adequately test the user's knowledge, and MCQ questions can be answered by the user through blind guessing. So, we can try to add a quiz that includes fill-in-the-blanks and short-answer type questions.

## 4.4. Local Deployment of LLM

Current applications mainly process user-uploaded text by using OpenAI's API services, which leads to a proportional increase in cost as the length of the uploaded text increases. In order to reduce the long-term cost, we can consider deploying LLM locally. Although this requires some initial hardware investment, the cost will be significantly lower once the local deployment is complete compared to using OpenAI's services continuously. Meanwhile, we can collect user feedback to pre-train and optimise LLM to improve the performance and user experience of the application.

I have attempted to convert and quantise the llama2 [13] model for deployment on my Mac using the llama.cpp tool[8]. However, the context window of the llama2 model is limited to 4096 tokens, which makes it difficult to meet the application's need to process longer uploaded documents. In addition, the runtime of the llama2 model deployed on Mac is too long and the quality of the generated content has a significant gap compared to the GPT model. These challenges suggest that although local deployment is cost-effective, further technical improvements and optimisations are needed to reach commercial standards.

[8] https://github.com/ggerganov/llama.cpp

# 5. Conclusion

Overall, this project has successfully developed an application that helps users consolidate their knowledge through quizzes. The application not only generates MCQ quizzes flexibly for files of various formats and sizes but also provides personalised feedback based on the user's answers. In addition, the project explores the various ways in which LLM handles long texts and technologies such as speech recognition.

During the development of the project, we also encountered a series of challenges, such as how to handle files in various formats, effectively control the cost, improve the processing speed, and how to precisely adjust the quality of generated content. Although this application still needs to be improved, it does help students revise. It also opens up new paths for innovation in the field of educational technology, demonstrating the great potential of using advanced AI technologies to promote learning and knowledge consolidation.

# Reference


[1] T. Alsubait, B. Parsia, and U. Sattler, “Ontology-Based Multiple Choice Question Generation,” *KI - Kunstliche Intelligenz*, vol. 30, no. 2, 2016, doi: 10.1007/s13218-015-0405-9.

[2] A. Vaswani *et al.*, “Attention is all you need,” in *Advances in Neural Information Processing Systems*, 2017.

[3] J. Devlin, M. W. Chang, K. Lee, and K. Toutanova, “BERT: Pre-training of deep bidirectional transformers for language understanding,” in *NAACL HLT 2019 - 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies - Proceedings of the Conference*, 2019.

[4] S. A. Lakshmi, R. Saturi, A. Bharti, M. Avvari, and B. Bhavana, “Multiple Choice Question Generation Using BERT XL NET,” 2023.

[5] A. Radford, J. W. Kim, T. Xu, G. Brockman, C. McLeavey, and I. Sutskever, “Robust Speech Recognition via Large-Scale Weak Supervision,” Dec. 2022, [Online]. Available: http://arxiv.org/abs/2212.04356

[6] V. Panayotov, G. Chen, D. Povey, and S. Khudanpur, “LIBRISPEECH: AN ASR CORPUS BASED ON PUBLIC DOMAIN AUDIO BOOKS”, Accessed: Apr. 24, 2024. [Online]. Available: http://www.gutenberg.org

[7] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever, “Improving Language Understanding by Generative Pre-Training,” 2018. [Online]. Available: https://gluebenchmark.com/leaderboard

[8] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever, “Language Models are Unsupervised Multitask Learners,” *OpenAI Blog*, vol. 1, no. 8, 2019.

[9] T. B. Brown *et al.*, “Language models are few-shot learners,” in *Advances in Neural Information Processing Systems*, 2020.

[10] L. Ouyang *et al.*, “Training language models to follow instructions with human feedback,” in *Advances in Neural Information Processing Systems*, 2022.

[11] OpenAI, “GPT-4 Technical Report,” Mar. 2023, [Online]. Available: http://arxiv.org/abs/2303.08774

[12] H. Touvron *et al.*, “LLaMA: Open and Efficient Foundation Language Models,” Feb. 2023, [Online]. Available: http://arxiv.org/abs/2302.13971

[13] H. Touvron *et al.*, “Llama 2: Open Foundation and Fine-Tuned Chat Models,” Jul. 2023, [Online]. Available: http://arxiv.org/abs/2307.09288

[14] Kurdi Ghader, Leo Jared, Parsia Bijan, Sattler Uli, and Al-Emari Salam, “A Systematic Review of Automatic Question Generation for Educational Purposes,” *Int J Artif Intell Educ*, vol. 30, no. 1, 2020, doi: 10.1007/s40593-019-00186-y.

[15] Y. Gao *et al.*, “Retrieval-Augmented Generation for Large Language Models: A Survey”, Accessed: Apr. 18, 2024. [Online]. Available: https://github.com/Tongji-KGLLM/

[16] H. Su *et al.*, “One Embedder, Any Task: Instruction-Finetuned Text Embeddings,” pp. 1102–1121.