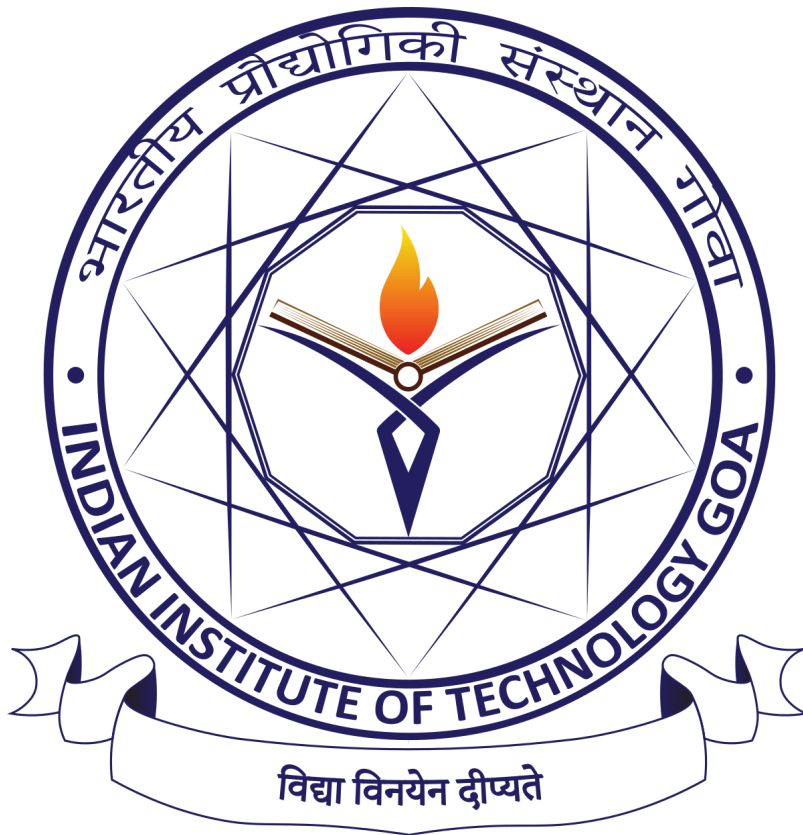


7th Semester
B.Tech Project Report
AcadBot2.0 an Q&A Application using LLM



Under Supervision

Dr. saurabh Trivedi

Team Members

Deepak Kumar

2103308

Hemant Singh Sidar

2103310

Mathematics & Computing

B. Tech

Acknowledgment

I would like to express my heartfelt gratitude to Dr. Saurabh Trivedi, from the School of Mathematics and Computer Science, IIT Goa, for his invaluable guidance, insightful feedback, and unwavering support throughout the course of my B.Tech project. His expertise and mentorship played a pivotal role in shaping the direction of my research and ensuring the successful completion of this project.

I am also deeply appreciative of everyone who provided insights, feedback, and encouragement during the course of this study. Your collective contributions have greatly enriched this project and enhanced its quality.

Dr. Trivedi's patience, dedication, and profound understanding of the subject have left a lasting impact on my academic journey, contributing not only to the success of this project but also to my overall growth as a learner. I am sincerely thankful for his mentorship and support.

ABSTRACT

This project focuses on the development of a **Question-and-Answer (Q&A) Application** that leverages **Natural Language Processing (NLP)** and machine learning techniques to deliver accurate and contextually relevant answers. The application enables users to interact through a chatbot interface, ask questions, and receive answers based on uploaded documents or stored data. It integrates modern AI technologies such as **Cohere's language generation API** and **SentenceTransformer embeddings** for document comprehension and contextual understanding.

The system architecture comprises a **Flask-based backend** that facilitates user authentication, data storage, and session management, ensuring secure and personalized interactions. Preprocessing and chunking of document data enhance the application's efficiency, enabling the retrieval of relevant information segments using semantic similarity metrics like cosine similarity. Additionally, the application supports session-based chat functionality, allowing users to track their queries and responses during active sessions.

This report presents the technical implementation details, including embedding preparation, similarity computation, and the use of enhanced prompts for precise answers. By integrating **state-of-the-art embedding models** and **language generation capabilities**, the project demonstrates an effective solution for document-based question answering while emphasizing modularity, scalability, and user-centric design.

TABLE OF CONTENTS

1. Introduction

1.1 Background of the Project

1.2 Problem Statement

1.3 Objectives

1.4 Scope of the Project

2. Literature Review

2.1 Overview of Question-Answering Systems

2.2 Document-Based Q&A Systems

2.3 Existing Approaches and Frameworks

2.4 Tools and Technologies

2.5 Integration of Web and Mobile Applications

3. System Design

3.1 Architecture Overview

3.2 Module Descriptions

- 3.2.1 Data Preparation and Storage
- 3.2.2 Embedding Generation
- 3.2.3 Query Processing
- 3.2.4 Chat Interface

4. Methodology

- 4.1 Data Collection and Preprocessing
- 4.2 Embedding Generation
- 4.3 Query Embedding and Similarity Search
- 4.4 Enhanced Prompt Creation
- 4.5 Response Generation
- 4.6 Frontend Implementation
- 4.5 Backend And Database Integration
- 4.6 Response Handling
- 4.9 API And Real-Time Interaction

5. Implementation

5.1 Dynamic Embedding Generator for Document-Based Chatbot

- 5.1.1 Document Loading
- 5.1.2 Text Chunking
- 5.1.3 Embedding Generation

Saving Embeddings and Chunks:

5.2 Document-Based Chatbot Using Precomputed Embeddings and Cohere API

- 5.2.1 Precomputed Embeddings and Chunk Management
- 5.2.2 Cohere API Integration
- 5.2.3 Embedding Model
- 5.2.4 Query Processing and Context Retrieval
- 5.2.5 Prompt Construction
- 5.2.6 Response Generation

6. Results and Discussion

6.1 Sample Screenshots of Application

- 6.1.1 Login and Registration
- 6.1.2 Data Storage and Retrieval
- 6.1.3 Chatbot Responses

7. Conclusion and Future Work

7.1 Summary of Achievements

7.2 Applications and Real-world Use Cases

7.3 Potential Enhancements

8. References

- Research Papers and Articles
- Python Libraries (Flask, SQLAlchemy, sentence-transformers)
- APIs (Cohere)

LIST OF FIGURES

LIST OF TABLES

CHAPTERS 1: Introduction

Efficient access to accurate and relevant information is a critical requirement in academic institutions, where students, faculty, and visitors often seek answers to a variety of questions. This report presents the design, development, and implementation of a Q&A application tailored to answer queries specifically related to IIT Goa, based on domain-specific documents. The application serves as both a web-based and mobile solution, providing a user-friendly interface and high-quality responses through the integration of advanced Natural Language Processing (NLP) technologies.

1.1 Background of the Project

Academic institutions like IIT Goa house a wealth of information spanning diverse topics, including academic programs, events, campus facilities, administrative processes, and more. This information is often scattered across various sources, such as websites, brochures, and notices, making it challenging for users to retrieve specific answers quickly.

To address this issue, the project leverages the capabilities of NLP to create a centralized, document-based Q&A system. By integrating pre-trained language models and embedding techniques, the application ensures accurate query processing and contextually relevant answers, all drawn directly from a curated set of documents related to IIT Goa.

1.2 Problem Statement

Traditional methods of accessing information, such as manual browsing or FAQ pages, are often inefficient and time-consuming, especially when dealing with large volumes of unstructured data. Users may struggle to find precise answers to specific queries, leading to frustration and inefficiency.

The primary challenge lies in developing a solution that can:

1. Handle a wide variety of user queries with high accuracy.
2. Ensure responses are contextually relevant and derived exclusively from predefined institutional documents.
3. Provide a seamless user experience across both web and mobile platforms.

1.3 Objectives

The main objectives of this project are:

- **To develop a centralized Q&A system** that provides accurate answers based on institutional documents.
- **To integrate advanced NLP techniques**, such as document chunking, embeddings, and query similarity, for precise response generation.
- **To ensure accessibility** through the creation of a web application and a mobile app with consistent functionality.
- **To maintain data security** by implementing user authentication and secure database management.
- **To enhance user experience** with an intuitive chat interface and fast response times.

1.4 Scope of the Project

This project focuses on building a scalable, domain-specific Q&A application for IIT Goa, with the following key features and limitations:

- The system is restricted to answering questions based on the provided documents, ensuring high accuracy within a defined scope.
- Advanced NLP models, such as Cohere's language models, are employed for query processing, with provisions for integrating additional models in the future.
- The application includes user-friendly interfaces for both web and mobile platforms, making it accessible to a broad audience.
- While the current implementation caters to IIT Goa, the system's design allows for easy adaptation to other domains by updating the document base.

This introduction sets the stage for understanding the motivation, technical foundation, and impact of the Q&A system, paving the way for a detailed exploration of its design, implementation, and results in subsequent sections.

CHAPTERS 2: Literature Review

This section examines the theoretical and practical underpinnings of Question-Answering (Q&A) systems, focusing on their evolution, methodologies, and application to educational institutions. It also highlights the technologies and frameworks that informed the design of this project and identifies the gaps in existing systems that the project addresses.

2.1 Overview of Q&A Systems

Question-Answering (Q&A) systems are designed to provide direct answers to user queries based on available data, improving the efficiency of information retrieval. Unlike traditional search engines that return a list of relevant documents, Q&A systems focus on delivering precise, context-aware answers.

- **Information Retrieval-Based Q&A:** This method retrieves answers by searching a large database or collection of documents for relevant information based on the user's query. The answer is directly fetched from existing data, and no new information is generated.
- **Knowledge-Based Q&A:** This approach uses structured knowledge bases, such as ontologies or databases, to provide precise answers. It often involves reasoning or querying a knowledge graph to extract accurate and contextually relevant information.
- **Generative Q&A:** In this method, answers are generated from scratch using models like GPT or other language models. Instead of retrieving direct answers from data, the system produces a response based on its understanding of the query and the knowledge it has been trained on.

Recent advancements in NLP and machine learning have significantly improved the accuracy and contextual understanding of Q&A systems, enhancing their effectiveness across various domains.

2.2 Document-Based Q&A Systems

- Document-based Q&A systems are designed to answer questions by retrieving information from a set of documents. These systems are commonly used in places like libraries, administrative offices, and academic departments to help users find information about policies, courses, and other resources.
- The system works by first processing the documents to organize the information. When a user asks a question, the system analyzes the query to understand what is being asked. It then searches the documents for the most relevant information. Using techniques like breaking down text (tokenization), creating text representations (embedding generation), and finding similarities between the query and documents, the system is able to identify and extract the right answer.
- For this project, the system is customized to work with IIT Goa's documents, allowing it to provide accurate answers to questions specifically about the institution.

2.3 Existing Approaches and Frameworks

2.3.1 What is Flask?

Flask is a lightweight Python web framework that is easy to set up and flexible. Flask is ideal for building simple web applications, APIs, and microservices due to its modularity and simplicity.

2.3.1.1 Flask Over Other Frameworks(like Django)

- It is lightweight and easy to configure.
- Offers flexibility in structuring the application.
- Simpler and more manageable for small projects.

2.3.1.2 Use of Flask

In this project, Flask was used to:

- Serve HTML templates for user registration, login, and chat pages.
- Handle form submissions for user login, registration, and question input.
- Manage sessions for storing user-specific chat history.
- Interact with the SQLite database through SQLAlchemy ORM(Object-Relational Mapping).



2.3.2 Large Language Models

Large language models (LLMs) are a type of foundational model trained on vast datasets, enabling them to understand and generate natural language as well as other types of content. Their capabilities allow them to perform a diverse array of tasks, making them valuable tools across various applications.

LLMs, as foundation models, are trained on massive datasets, allowing them to support multiple use cases and applications without the need for domain-specific models. This approach reduces costs, streamlines infrastructure, and often results in superior performance compared to training separate models for individual tasks.

Notable LLMs include OpenAI's GPT-3 and GPT-4, supported by Microsoft, Meta's Llama models, Google's BERT, RoBERTa, and PaLM, as well as IBM's Granite models integrated

into watsonx.ai. LLMs are designed to generate human-like text and other content, leveraging billions of parameters to understand context, produce relevant responses, translate languages, summarize information, answer questions, and assist in creative and coding tasks.

These models have revolutionized applications such as chatbots, virtual assistants, content generation, research tools, and language translation. As LLMs evolve, they are set to redefine interactions with technology and information access, solidifying their role in the modern digital landscape.

2.3.2.1 How large language models work

LLMs leverage deep learning and vast textual data, typically built on a transformer architecture like the generative pre-trained transformer (GPT), which excels at processing sequential text data. These models consist of multiple neural network layers, fine-tuned through training and supported by an attention mechanism that focuses on specific data segments to improve context understanding.

During training, LLMs learn to predict the next word in a sentence by analyzing the context provided by preceding words. This involves assigning probability scores to tokenized text—text broken into smaller character sequences—and converting these tokens into embeddings, numerical representations that capture contextual meaning.

Training on massive text corpora, often encompassing billions of pages, allows LLMs to learn grammar, semantics, and conceptual relationships through zero-shot and self-supervised learning. Once trained, they can generate coherent, contextually relevant text by predicting the next word based on input, drawing from the patterns they have learned. This ability supports a range of natural language understanding (NLU) and content generation tasks.

To improve model performance and mitigate issues like biases, inaccurate outputs (hallucinations), and inappropriate content, strategies such as prompt engineering, prompt-tuning, fine-tuning, and reinforcement learning with human feedback (RLHF) are used. These approaches are essential for ensuring LLMs meet enterprise-grade standards and do not pose liability or reputational risks to organizations.

2.3.2.2 Real World applications of LLMs

LLMs are redefining business processes across industries, showcasing their versatility in various tasks and use cases. They enhance conversational AI in chatbots and virtual assistants (e.g., IBM watsonx Assistant and Google's BARD), delivering context-aware, human-like responses that elevate customer service interactions.

In content generation, LLMs automate the creation of blog articles, marketing materials, and other written content. In research and academia, they help summarize and extract insights from large datasets, speeding up knowledge discovery. LLMs also play a crucial role in language translation, enabling accurate and contextually appropriate multilingual communication. Additionally, they assist in coding by generating and analyzing code and even converting code between different programming languages.

LLMs contribute to accessibility by supporting applications like text-to-speech and producing content in formats that aid individuals with disabilities. Industries from healthcare to finance leverage LLMs to streamline processes, improve customer experiences, and enable data-driven decision-making.



Key areas where LLMs benefit organizations include:

- **Text generation:** Producing language outputs such as emails, articles, and reports in response to prompts. This capability includes advanced techniques like retrieval-augmented generation (RAG).
- **Content summarization :** Condensing lengthy articles, news, reports, corporate documents, and customer histories into concise formats tailored to the output needs.
- **AI assistants:** Powering chatbots that handle customer inquiries, complete backend tasks, and provide detailed responses, forming an integral part of self-service customer care.
- **Code generation:** Assisting developers in coding tasks, identifying bugs, improving security, and translating code between languages.
- **Sentiment analysis:** Evaluating text to understand customer emotions, facilitating brand reputation management and feedback analysis.
- **Language translation :** Enabling organizations to engage with global audiences through fluent, context-aware translations and multilingual capabilities.

LLMs can transform every sector from finance and insurance to HR and healthcare by automating customer service, improving response times, and enhancing accuracy in context gathering. With simple API integration, they offer accessible and impactful solutions for modern businesses.

Ref1 : cotext

Ref2: Image1

Ref3:Image2

2.3.3 Introduction of LlamaIndex:

LlamaIndex is a comprehensive framework designed for building context-augmented generative AI applications with LLMs, enabling the development of agents and customized workflows. The platform caters to a wide range of users, from beginners to advanced developers.

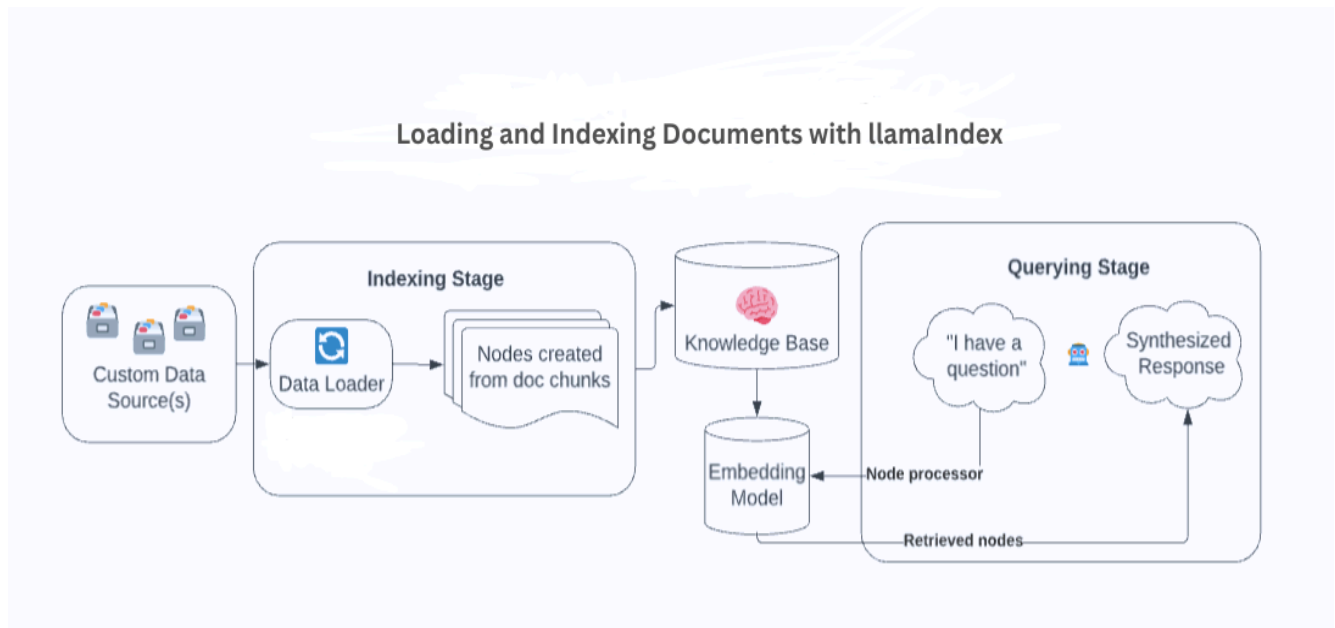
For beginner users, LlamaIndex offers a high-level API that simplifies data ingestion and querying, allowing them to interact with their data using just five lines of code. Advanced users benefit from lower-level APIs that provide extensive customization options. These APIs enable users to modify and extend various components such as data connectors, indices, retrievers, query engines, and reranking modules to meet specific application requirements.

Additionally, LlamaIndex supports the development of agents, including the use of Retrieval-Augmented Generation (RAG) pipelines as one of the tools to facilitate task completion. This flexibility makes LlamaIndex an effective platform for creating tailored AI-driven solutions.

2.3.3.1 Use Cases of LlamaIndex:

LlamaIndex and context augmentation offer a wide range of use cases, including:

- **Question-Answering (Retrieval-Augmented Generation, RAG):** Enhancing LLMs with external data sources for precise answers.
- **Chatbots:** Developing interactive, intelligent chat applications.
- **Document Understanding and Data Extraction:** Analyzing and extracting information from unstructured documents.
- **Autonomous Agents:** Creating agents capable of conducting research and executing actions autonomously.
- **Multi-modal Applications:** Integrating and processing text, images, and other data types in a cohesive manner.
- **Model Fine-Tuning:** Customizing models using specific data to improve their performance on targeted tasks.



Ref:[llamaindex](https://llamaindex.ai)

2.3.4 Retrieval-Augmented Generation (RAG)

Retrieval-Augmented Generation (RAG) combines information retrieval with text generation to enhance the performance of language models. Here's how it works:

- **Retrieval** refers to the process of retrieving relevant information from a knowledge base.
- **Augmented** means this retrieved information is used to assist the model.
- **Generation** involves the model generating text based on the retrieved data.

At its core, RAG aims to improve the model's output by incorporating external knowledge, ensuring that the generated responses are more accurate and contextually relevant.

A fascinating aspect of RAG is that the language model itself has never "seen" the exact data it processes. Models across various domains—such as computer vision and speech recognition—do not directly understand raw inputs like images, audio, or text. Instead, these inputs are converted into mathematical representations. For instance, text is transformed into embeddings, images into matrices, and audio into spectrograms. This allows models to process and understand data in a numerical format.

Here's how RAG connects a language model to an external knowledge base:

1. **Chunked Knowledge Representation:** Information is broken down into smaller units, called chunks, which can be sentences, paragraphs, or specific concepts within the text.
2. **Embeddings (Capturing Meaning in Numbers):** Each chunk is then converted into an embedding—a numerical representation that captures the key information of the chunk.
3. **Storing the Embeddings:** These embeddings are stored in a database, creating a searchable knowledge base.
4. **Retrieval:** When a user submits a query, it is also converted into an embedding. The system then retrieves the most semantically similar chunks from the database.
5. **Generation:** The retrieved information, along with the user query, is fed into the language model, which generates a more informed and relevant response.

2.4 Technologies and Tools

- **Backend Framework:**
 - **Flask:** A lightweight web framework used to handle HTTP requests, render templates, and manage application routing.
- **Database Management:**
 - **SQLAlchemy:** An Object-Relational Mapping (ORM) tool for managing database interactions.
 - **Flask-Migrate:** Facilitates database migrations, ensuring the database schema remains consistent with the application models.
 - **SQLite:** Used as the default database for development purposes, with the flexibility to switch to other SQL databases as needed.
- **Artificial Intelligence and Natural Language Processing:**
 - **Sentence-Transformers:** Utilized for generating text embeddings, enabling efficient similarity searches.
 - **Cohere API:** Provides advanced language model capabilities for generating coherent and contextually accurate responses.
 - **NLTK (Natural Language Toolkit):** Assists in text preprocessing tasks such as tokenization and lemmatization.
- **Data Processing and Storage:**
 - **NumPy & Pickle:** Handle numerical operations and serialization of data structures, respectively.
 - **Llama Index:** Facilitates document loading and management from specified directories.
- **Deployment Tools:**
 - **WSGI (Web Server Gateway Interface):** Ensures the Flask application can be deployed on various web servers.
 - **Run Scripts:** Separate scripts (wsgi.py and run.py) manage different deployment scenarios and environments.

2.5 Integration of Web and Mobile Applications

Integration of web and mobile applications has become a critical aspect of modern software systems to ensure accessibility and usability across different platforms. For a question-answering (QA) system, this integration allows users, including students and faculty, to access information seamlessly, whether through a browser or a mobile app.

Key Benefits

- **Cross-Platform Accessibility:** Ensures that users can interact with the QA system from multiple devices, enhancing convenience.
- **Consistent User Experience:** Provides a unified design and functionality, ensuring smooth transitions between web and mobile interfaces.
- **Broader Reach:** Accommodates users who prefer mobile apps over web platforms, thus increasing system adoption.

Implementations

- **Backend:**
 - A centralized Flask-based backend handles both web and mobile application requests, ensuring consistent data processing and API responses.
- **Frontend:**
 - The web interface is designed using HTML and CSS for browsers.
 - The mobile app, built using React Native (or equivalent), communicates with the Flask API to provide similar functionality on handheld devices.
- **Data Synchronization:**
 - A shared database ensures that user queries, answers, and document updates are synchronized across platforms in real time.

This integration enhances the accessibility of institutional information related to IIT Goa, catering to diverse user preferences while maintaining functionality and reliability.

CHAPTERS 3: System Design

The system design section provides an in-depth look at how the project is structured, its functional components, and the interactions between various modules. Below is a detailed breakdown:

3.1 Architecture Overview

The system employs a modular, service-oriented architecture to enable efficient document-based question answering across web and mobile platforms. It features a responsive frontend for the web (HTML and CSS) and mobile (React Native), a Flask-based API for user authentication, query processing, and document retrieval, and an SQLite database for managing user data, document metadata, embeddings, and chat histories. The NLP engine leverages pre-trained Hugging Face models and LlamaIndex for embedding generation, query processing, and answer retrieval. The embedding engine facilitates semantic search through text chunking and embedding generation. The application is production-ready, utilizing WSGI (Gunicorn) for serving and Docker for containerization.

3.2 Module Descriptions

3.2.1 Data Preparation and Storage

- **Functionality:** Prepares the dataset for use in the system by chunking documents and generating embeddings.
- **Implementation Details:**
 - Documents are read using the SimpleDirectoryReader module from LlamaIndex.
 - Text is chunked into manageable units, such as sentences or paragraphs, using NLTK.
 - Each chunk is embedded using the Hugging Face all-mpnet-base-v2 model.
- **Storage:**
 - Chunk embeddings are saved as NumPy arrays for fast retrieval.
 - Chunks themselves are serialized and stored using pickle.

3.2.2 Embedding Generation

- **Functionality**
 - Generates numerical representations (embeddings) of text for semantic similarity comparison.
- **Implementation Details**
 - The all-mpnet-base-v2 model from Hugging Face SentenceTransformers library is used.
 - Each document chunk is passed through the model to generate a high-dimensional vector.
 - These vectors are saved to disk for future query processing.
- **Advantages**
 - Enables fast and accurate similarity searches.
 - Reduces processing time during query execution.

3.2.3 Query Processing

- **Functionality**
 - Matches user queries against document embeddings to identify relevant information.
- **Implementation Details**
 - User input is embedded using the same pre-trained model.
 - Cosine similarity is calculated between the query embedding and document embeddings.
 - The top-ranked chunks are retrieved and processed to form a meaningful response.
- **Techniques**
 - **Semantic Search:** Finds document chunks with the highest relevance to the query.
 - **Thresholding:** Ensures that only chunks with similarity scores above a set threshold are considered.

3.2.4 Frontend Development

- **Functionality:** Provides a user-friendly interface for interaction with the system.
- **Implementation Details**
 - Web Interface:
 - Built using HTML and CSS.
 - Mobile Interface:
 - Developed using React Native.
 - Provides similar functionality as the web interface.
 - Features:
 - Login and registration forms for user authentication.
 - Real-time chat interface for entering questions and receiving answers.
- **Web Pages:**
 - **Authentication Page:** The website includes user authentication pages for login and registration. These pages allow users to securely register and log in to the system.
 - **Login Page:** Users input their username and password, which are validated against the database. If successful, they are redirected to the main page.
 - **Registration Page:** New users register by providing their name, username, email (validated to be a gmail.com address), and password.
 - **Main Page and Data Storage:** After logging in, users can store data through the main page. This data is saved in the Data_Table and displayed on the success page. Also this page includes the navigation button to move to the chat interface.
 - **Chat Interface:** The chat interface allows users to input questions and receive pre-set answers (currently "Working on it."). Each question is stored in the Question table, ensuring session-based question tracking. If a question is repeated in the same session, the existing answer is fetched and displayed.

CHAPTER 4 : METHODOLOGY

The chatbot methodology focuses on building a system to retrieve and generate accurate answers from documents. Documents are processed by cleaning and dividing them into smaller chunks, which are then converted into numerical embeddings using a pre-trained model. User queries are also transformed into embeddings and matched with document embeddings to identify relevant information. This information is used to create a prompt for a language model, like Cohere, to generate precise responses. The chatbot ensures efficient, real-time interaction by using precomputed embeddings and retrieving only relevant data.

4.1 Data Collection and Preprocessing

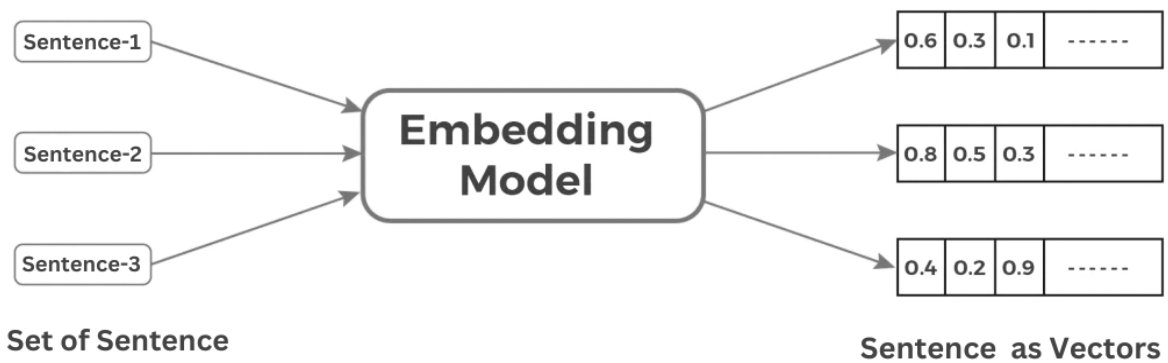
The data used for the chatbot is sourced from the official IIT Goa website and its Wikipedia page. The chatbot uses data stored in the `data` directory, which are processed to enable efficient information retrieval. The main preprocessing steps include:

- **Text Chunking:** Documents are split into smaller chunks, usually by paragraphs, to improve similarity calculations. Sentence-level chunking can also be used for more detailed processing.
- **Preprocessing Tools:** The NLTK library is used for tasks like tokenization and segmentation, ensuring consistent and accurate chunking of the text.

4.2 Embedding Generation

The chatbot uses a pre-trained embedding model to convert text chunks into numerical vectors.

- **Model Selection:**
The chatbot employs the `sentence-transformers/all-mpnet-base-v2` model, renowned for its performance in semantic similarity tasks.
- **Text-to-Vector Conversion:**
Each chunk is transformed into a dense vector embedding that encapsulates its semantic meaning. This step enables the chatbot to compare chunks effectively and retrieve contextually relevant information.

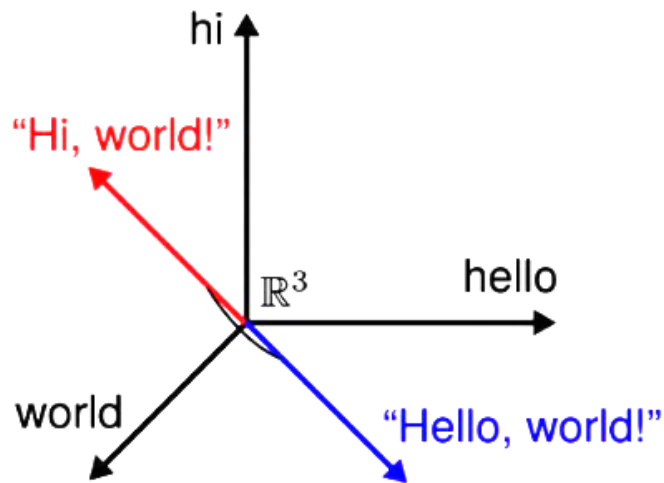


- **Efficient Storage:** Embeddings are saved using NumPy and Pickle, ensuring that only new or updated files are reprocessed, improving efficiency.

4.3 Query Embedding and Similarity Search

User queries are processed and compared with document chunks.

- **Query Embedding:** The user's query is converted into a vector using the same embedding model, ensuring that both queries and document chunks are represented consistently.
- **Similarity Calculation:** The chatbot compares the query with the pre-computed document embeddings using different similarity measures:
 - **Cosine Similarity (default):** Measures the angle between vectors, making it effective for text-based data where the direction of the vector matters more than its size.



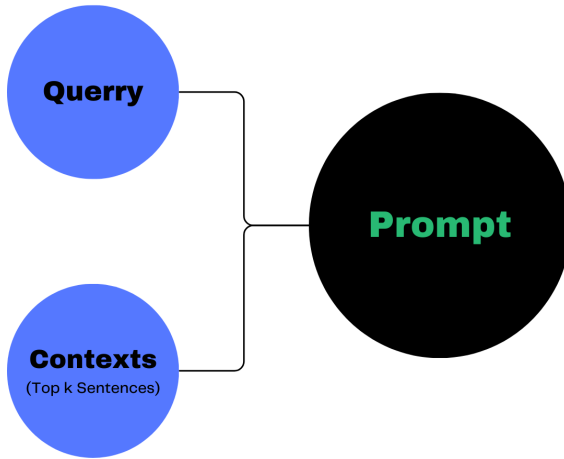
Cosine Similarity

- **Alternative Metrics:** Other options like Euclidean distance and inner product are also available, offering flexibility for different use cases.
- **Top-N Retrieval:** The system selects the top N most relevant chunks based on similarity scores and uses them to generate the response, ensuring the chatbot provides answers grounded in the most relevant information.

4.4 Enhanced Prompt Creation

An enhanced prompt is created to help the language model generate accurate and relevant responses. It combines the following elements:

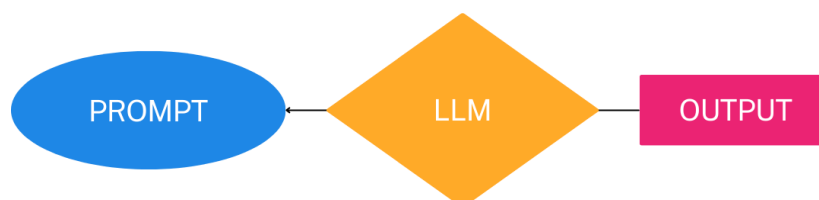
- **Top Chunks:** The most relevant document chunks, selected through the similarity search, to provide the necessary context for the user's question.
- **User Query:** The user's original question, included in the prompt to ensure the response directly answers their needs.



4.5 Response Generation

The chatbot uses a powerful language model from Cohere to generate responses based on the enhanced prompt. The chatbot integrates the Cohere API to generate high-quality responses. The process involves the following steps:

- **Model Utilization:** The Cohere command-xlarge-nightly model is used for its ability to produce coherent, contextually accurate, and detailed answers.
- **Prompt Construction:** A structured prompt is created by combining relevant document chunks and the user's query. This ensures the model has sufficient context to generate precise responses.
- **Parameter Configuration:** Specific API parameters are set to optimize performance:
 - **Temperature:** A low value (0.2) is used to produce consistent and deterministic outputs.
 - **Token Limit:** A maximum of 100 tokens is specified to maintain concise and focused responses.
- **Response Generation:** The enhanced prompt is sent to the Cohere API, and the model's output is delivered as the chatbot's response to the user.



4.6 Frontend Implementation

The frontend of the chatbot was developed using HTML, CSS, and Flask templates to provide a user-friendly interface for interaction. The website allows users to authenticate, store data, and engage with the chatbot. The following steps outline the key frontend components:

- **Authentication:** The login and registration pages allow users to authenticate before interacting with the chatbot. Passwords are securely handled using hashed storage, ensuring that sensitive user data is protected. A simple form-based authentication mechanism was implemented, as seen in the login.html and register.html templates. On successful login, users are redirected to the main page, where they can store data or chat.
- **Data Storage:** The main.html page provides a text input box where users can enter data to be stored in the database. This feature is essential for building a personalized knowledge base for the chatbot. Users can type any content into the input field, and the data is stored in the Data_Table model associated with their account. The stored data is saved in an SQLite database, which is then available for future reference during chatbot interactions.
- **Chat Interface:** The chat.html template allows users to ask questions to the chatbot. The questions and corresponding responses are dynamically displayed on the page using a Flask loop to render previous interactions stored in the database. This ensures that the chat history is preserved and presented to the user in a clear and organized manner.
- **Session Management:** The application uses session IDs to maintain the state of each user's conversation with the chatbot. Each user interaction is tied to a unique session, which allows for personalized query-answer sessions. This is managed through Flask's session management mechanism.

4.7 Backend and Database Integration

The backend of the chatbot is implemented using Flask, with SQLAlchemy used for database integration and migration. The key components of the backend are:

- **Database Models:** The User, Question, and Data_Table models are central to the operation of the chatbot. The User model handles user authentication, storing usernames, passwords (hashed), and other details. The Question model stores

questions and answers during a session, while the `Data_Table` model holds user-specific data that can be referenced during chatbot interactions.

- **Database Interaction:** The database interaction is handled through SQLAlchemy ORM, which abstracts the database queries and allows seamless interaction with the SQLite database. The `db.session` object is used to commit changes to the database, whether adding new questions, updating answers, or storing user data.
- **Routes and Logic:** Several Flask routes are defined to handle different actions in the application. The `register` and `login` routes manage user authentication, while the `store` route allows for the storage of user data. The `/chat` route handles the interaction with the chatbot, including storing questions and fetching responses from the database.

4.8 Response Handling

Once a user submits a question, the system processes it in the following steps:

1. **Query Processing:** The chatbot receives the user's query through a form submission, which is handled by the `chat()` function in Flask. If the query does not already exist in the database, a new `Question` object is created, and the chatbot's fixed response ("Working on it.") is stored.
2. **Database Interaction:** The question is either stored as a new entry in the `Question` table or updated if it already exists. The chatbot's answer is stored alongside the question, and the interaction is saved in the session-specific context.
3. **Enhanced Prompt Creation:** The stored question and answer are used as part of the prompt for the language model, ensuring that the response generated is contextually relevant. The question is paired with the appropriate document chunks based on the similarity search to provide accurate and relevant answers.
4. **Response Generation:** After processing the query and fetching the relevant document chunks, the chatbot uses the Cohere language model to generate a response. The response is then displayed in the chat interface, where users can continue their interaction.

4.9 API and Real-Time Interaction

The application provides a real-time chat experience by utilizing Flask's form handling and session management. When a user submits a question, it triggers an immediate response. The interface is designed to display the conversation history dynamically, ensuring a smooth and engaging user experience.

CHAPTER 5: IMPLEMENTATION

Dynamic Embedding Generator for Document-Based Chatbot

The system processes documents dynamically from a directory, generates embeddings for document chunks, and saves them for efficient retrieval and query processing. This approach ensures the chatbot adapts to new data seamlessly while maintaining performance.

1. Document Loading

The system reads documents from the specified directory (`data`) using the `SimpleDirectoryReader` from `LlamaIndex`. Each document's content is processed to generate embeddings.

```
# Load documents from the "data" directory
from llama_index.core import SimpleDirectoryReader
documents = SimpleDirectoryReader("data").load_data()
```

2. Text Chunking

Documents are split into manageable chunks to enable granular embedding generation and efficient query processing. Two chunking methods are supported:

- **Sentence-level** using `nlk.sent_tokenize`.
- **Paragraph-level** using `text.split("\n\n")`


```
def chunk_document(text, chunk_by='paragraph'):
    if chunk_by == 'sentence':
        return nltk.sent_tokenize(text)
    elif chunk_by == 'paragraph':
        return text.split("\n\n")
    else:
        raise ValueError(f"Unsupported chunking method: {chunk_by}")
```

3. Embedding Generation

The `all-mpnet-base-v2` model from `sentence-transformers` is used to generate embeddings for each chunk. The generated embeddings enable efficient similarity calculations for query responses.

```
embedding_model = SentenceTransformer('sentence-transformers/all-mpnet-base-v2')

def get_text_embedding(text):
    return embedding_model.encode(text, convert_to_tensor=False)
```

Saving Embeddings and Chunks:

The embeddings and their corresponding chunks are saved to disk for future use. This allows the system to skip the embedding generation step in subsequent runs, enhancing efficiency.

```

chunks = []
chunk_embeddings = []
for doc in documents:
    doc_chunks = chunk_document(doc.text, chunk_by='paragraph')
    chunks.extend(doc_chunks)
    for chunk in doc_chunks:
        chunk_embeddings.append(get_text_embedding(chunk))

chunk_embeddings = np.array(chunk_embeddings)

# Save the embeddings and chunks for future use
print("Saving embeddings and chunks...")
np.save(embedding_file, chunk_embeddings)
with open(chunks_file, 'wb') as f:
    pickle.dump(chunks, f)

```

Document-Based Chatbot Using Precomputed Embeddings and Cohere API

1. Precomputed Embeddings and Chunk Management

The chatbot system uses precomputed document embeddings and their corresponding text chunks stored in files for faster query responses. These files are loaded at runtime if they exist.

```

# Set file paths for loading embeddings
embedding_file = "saved_embeddings.npy"
chunks_file = "saved_chunks.pkl"

# Load embeddings and chunks
if os.path.exists(embedding_file) and os.path.exists(chunks_file):
    chunk_embeddings = np.load(embedding_file)
    with open(chunks_file, 'rb') as f:
        chunks = pickle.load(f)
else:
    raise FileNotFoundError("Embeddings or chunks file not found. Run `prepare_embeddings.py` first to generate them.")

```

2. Cohere API Integration

The Cohere API is used to generate answers to user queries based on the retrieved

context. The chatbot uses the `command-xlarge-nightly` model, configured with specific parameters for generating concise and structured responses.

```
# Initialize Cohere client (or other language model API)
import cohere

api_key = 'cxqcLuiWORNXmgZSKdiJEvtmCvhH7NZsFixUwlp' # Replace with your Cohere API key
co = cohere.Client(api_key)
```

3. Embedding Model

The chatbot uses the `all-mpnet-base-v2` model from `sentence-transformers` to generate embeddings for user queries. This ensures accurate semantic similarity computation between user input and document chunks.

```
# Set the embedding model to a more powerful one from sentence-transformers
embedding_model = SentenceTransformer('sentence-transformers/all-mpnet-base-v2', device="cpu")

def get_text_embedding(text):
    return embedding_model.encode(text, convert_to_tensor=False)
```

4. Query Processing and Context Retrieval

The system computes similarities between the user's query embedding and the precomputed document embeddings using various similarity metrics (`cosine`, `euclidean`, or `inner_product`). The top N most relevant chunks are retrieved to construct the context.

```
# Similarity metrics: Cosine, Euclidean, or Inner Product
def compute_similarity(query_embedding, document_embeddings, metric='cosine'):
    if metric == 'cosine':
        similarities = np.dot(document_embeddings, query_embedding) / (
            np.linalg.norm(document_embeddings, axis=1) * np.linalg.norm(query_embedding)
        )
    elif metric == 'euclidean':
        similarities = -np.linalg.norm(document_embeddings - query_embedding, axis=1) # Negative for sorting
    elif metric == 'inner_product':
        similarities = np.dot(document_embeddings, query_embedding)
    else:
        raise ValueError(f"Unsupported similarity metric: {metric}")

    return similarities
```

5. Prompt Construction

An enhanced prompt is dynamically generated to guide the Cohere API. It includes the retrieved context and the user query, instructing the model to generate a precise and relevant response.

```
# Improved prompt with specific instructions
def create_enhanced_prompt(context, query):
    enhanced_prompt = (
        f"Context: {context}\n\n"
        "Based on the context provided above, please provide a concise, accurate, and well-structured answer to the following query.\n\n"
        f"Query: {query}\n\n"
        "Please ensure the response is clear and directly addresses the query while utilizing relevant information from the context.\n\n"
        "Answer in a concise and precise manner:"
    )
    return enhanced_prompt
```

6. Response Generation

Once the user's query is processed, the chatbot generates a response by combining the retrieved context and user input into a structured prompt. This prompt is then passed to the Cohere API to produce an accurate and concise answer.

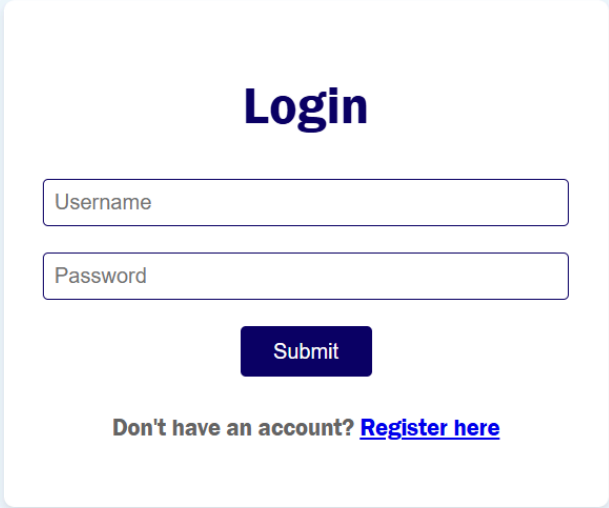
The enhanced prompt is sent to the Cohere API's `command-xlarge-nightly` model, configured to generate concise and structured answers. Parameters like `temperature` and `max_tokens` control the creativity and length of the response, respectively.

```
# Function to query Cohere API
def cohere_llm(prompt, temperature=0.2, max_tokens=100):
    response = co.generate(
        model='command-xlarge-nightly',
        prompt=prompt,
        max_tokens=max_tokens,
        temperature=temperature
    )
    return response.generations[0].text.strip()
```

CHAPTER 6: Results and Discussion

6.1 Sample Screenshots of Application

- 6.1.1 Login and Registration




A screenshot of a login form centered on a light blue background. The form is a white rounded rectangle with a subtle drop shadow. At the top, the word "Login" is displayed in a bold, dark blue font. Below the title are two input fields: the first is labeled "Username" and the second is labeled "Password", both in a light gray font. A dark blue "Submit" button is positioned below the password field. At the bottom of the form, the text "Don't have an account?" is followed by a blue, underlined link that says "Register here".

Register

Already have an account? [Login here](#)

- 6.1.2 Data Storage and Retrieval

[Logout](#)


AcadBot 2.0

Type here...

STORE DATA

[Have you any question?](#)

- 6.1.3 Chatbot Responses

[Logout](#)

AcadBot 2.0

Bot: The full form of IIT is the Indian Institute of Technology.

Bot: Prof. Dharendra S Katti is the director of IIT Goa.

Bot: The context provided does not specify the number of branches available in the B.Tech. program at IIT Goa. It only mentions that students can change their branch after completing their first two semesters.

You: What is the full form of IIT?

You: Who is director of IIT Goa?

You: How many branches are there in B.tech IIT Goa?

You: Do you have any Idea about M tech program at IIT Goa?

Your question

Submit

CHAPTER 7: Conclusion and Future Work

7.1 Summary of Achievements

This project is a Question-Answering (Q&A) System designed specifically for IIT Goa, aimed at answering user queries using information from official documents. The system is available as both a web-based application and a mobile app, ensuring easy access for users across platforms.

The project uses Natural Language Processing (NLP) techniques and advanced models to process documents and find accurate answers. It leverages tools like Hugging Face models and LlamaIndex for generating embeddings (representations of text) and matching user questions with relevant document content. The Sentence-Transformers/all-mpnet-base-v2 model is used for creating embeddings, and the system divides documents into smaller sections (chunks) for better search results.

The system includes:

- Frontend: A simple and user-friendly web interface (built with HTML and CSS) and a mobile app.
- Backend: A Flask-based server that handles login, queries, and data processing.
- Database: SQLite, used for saving user data, document information, and chat history.
- NLP Engine: Responsible for understanding queries and retrieving accurate answers from documents.

The application has been deployed using tools like Gunicorn and Docker to make it reliable and scalable.

Key features include:

- Secure user login.
- Accurate responses to questions using document-based search.
- Support for both web and mobile users.

This project bridges a gap in existing systems by focusing on IIT Goa-specific queries. It demonstrates how modern technology can make institutional information more accessible to students, staff, and others. It also highlights future possibilities like adding

more features and making the system scalable for larger audiences.

In summary, this project provides an easy-to-use and effective solution for answering questions related to IIT Goa, showcasing the practical use of modern technology in education.

7.2 Applications and Real-world Use Cases

The question-answering system developed as part of this project has several practical applications and real-world use cases, especially in the context of educational institutions like IIT Goa. Below are some of the key areas where this system can make a significant impact.

7.2.1 Applications

- **Educational Institutions**
 - **Quick Information Access:** Students, faculty, and visitors can easily find details about programs, processes, facilities, and events.
 - **Efficient Communication:** Reduces administrative workload by automating FAQs.
 - **Tailored Assistance:** Delivers specific answers for user queries.
- **Document Management**
 - **Easy Retrieval:** Quickly locates information from large document sets.
 - **Centralized Repository:** Acts as a knowledge base for institutional documents.
- **Web and Mobile Integration**
 - **Cross-Platform Access:** Works seamlessly on web and mobile platforms.
 - **Simplified Interaction:** User-friendly design makes data easily accessible.

7.2.2 Real-World Use Cases

- **University Portals:** Similar systems can be implemented in other universities and colleges to improve information dissemination and support services.
- **Corporate Knowledge Management:** Adaptable to industries and organizations to enable efficient query resolution from internal databases or documentation.
- **Government Services:** Can be used to provide citizens with quick answers to queries related to public policies, services, and procedures.

7.3 Potential Enhancements

1. Support for Additional Models

- Integrate advanced NLP models for improved accuracy and faster query resolution.
- Explore other state-of-the-art tools like Cohere or GPT-based models for enhanced performance.

2. Scalability Improvements

- Upgrade infrastructure to handle a larger volume of users and data efficiently.
- Implement distributed databases and cloud-based storage for better performance.

3. Multilingual Support

- Expand capabilities to process queries and documents in multiple languages for broader accessibility.

4. Voice-based Interaction

- Add support for voice queries to make the system more user-friendly, especially for mobile users.

5. Integration with External APIs

- Connect with third-party systems for real-time updates, such as college event schedules or academic results.

6. Enhanced Personalization

- Use AI to tailor responses based on user profiles, preferences, or frequently asked queries.

7. Improved Mobile Experience

- Optimize mobile app performance and UI for a smoother user experience.
- Add offline capabilities for accessing stored data without an active internet connection.

8. Real-Time Document Updates

- Automate the synchronization of new documents and ensure the system remains up-to-date.

9. Advanced Analytics and Reporting

- Implement dashboards to analyze user queries and system performance, helping improve service quality.

10. Integration with Smart Assistants

- Enable compatibility with devices like Alexa or Google Assistant for seamless interactions.

CHAPTER 8: References

- Research Papers and Articles
- Python Libraries (Flask, SQLAlchemy, sentence-transformers)
- APIs (Cohere)