Decentralized RAG: Combining Retrieval Augmented Generation with Blockchains

Reed White (12367576)

Week 3 Report

The following progress has been made during this sprint (weeks 2-3):

- Proof-of-Concept Smart Contract (DocumentStore.sol)
    - The “Corpus” contract is an extremely simple smart contract developed with Solidity that accepts arrays of chunks (documents split by the frontend) and IDs and stores them in the blockchain. Simplicity is being emphasized, because this proof-of-concept is attempting to keep the system as simple as possible by storing the corpus directly in the blockchain, which removes the need for an extra layer of dedicated, distributed, off-chain storage. However, this is potentially not scalable for production use by businesses, as it may be prohibitively expensive to store full-sized corpuses. It contains the following struct and methods:
        - Document{}: this is a struct object that contains an array of the text chunks of a submitted document, a matching array that contains IDs for each text chunk, a metadata string to describe the document, and an address to track which account submitted the document.
        - corpus[]: this is a dynamic storage array that stores Document{} objects.
        - insert(): this requires an array of text chunks, a metadata string, and an array of document IDs and stores them in the dynamic corpus[] array as a Document{}. Please note that these IDs are not provided by end users and are generated by the contract/frontend as every chunk needs a unique ID to be stored in a vector store.
        - retrieveDocument(): this requires a key and retrieves the related Document{} from corpus[].
        - retrieveLatestKey(): this requires no parameters and returns the latest index key for corpus[], based on the length of corpus[].
        - retrieveLatestID(): this requires no parameters and returns the latest ID of the latest chunk in Document{}, based on the length of corpus[] and the length of the ID array in the latest Document{}.
        - retrieveDocumentBySender(): this requires an address and returns a list of all indexes in corpus[] that contain Documents submitted by the provided address.
        - removeDocument(): this requires a key and removes the Document{} at the provided index key of corpus[].

        - removeDocumentBySender(): this requires an address and removes all Document{} objects stored in corpus[] submitted by the provided address.
        - constructor(): this requires no parameters and simply initializes corpus[] with a sample Document{}.
- Proof-of-Concept Web3.js Interaction Script (corpus_interact.js)
    - This script (corpus_interact.js) acts as a simple interface between the plain HTML/CSS frontend and the above Corpus smart contract. It uses Web3.js to facilitate the interactions and MetaMask to manage accounts/keys. It contains the following methods:
        - insertDocument(): this requires a text string, a metadata string, and an ID and calls the Corpus.insert() method to create a new Document with the provided details and append it to corpus[]. This inserts the provided string as a single chunk.
        - retrieveDocument(): this requires a key and calls the Corpus.retrieveDocument() method to retrieve and display the Document[] located at the index in corpus[] that matches the key (if found).
        - retrieveDocumentKeysBySender(): this requires an address and calls the Corpus.retrieveDocumentKeysBySender() method to retrieve the keys in corpus[] for all Document{} objects that were submitted by the provided address (if found).
        - retrieveLatestKey(): this requires no parameters and calls the Corpus.retrieveLatestKey() method to search corpus[] for the latest key that has a Document{}.
        - retrieveLatestID(): this requires no parameters and calls the Corpus.retrieveLatestID() method to search corpus[] for the latest chunk ID in the latest Document{} in corpus[].
        - inputFile(): this requires a text file and a metadata string. The text file is chunked by number of characters directly in the script and stored as an array. retrieveLatestID() is called to find the latest ID, and a matching array of unique IDs are generated to match the array of chunks. Corpus.insert() is then called to insert the array of chunks, the array of IDs, the metadata string, and the address of the sender as a single Document{} into corpus[].
        - displayFileContent(): this requires a text document and is initiated by the end user selecting a text document to upload. It displays the document in their browser.
        - readFileContent(): this requires a text document and is invoked by the inputFile() and displayFileContent() methods to read uploaded user documents.
        - removeDocument(): this requires a key and calls the Corpus.removeDocument() method to delete the document in corpus[] located at the index that matches the key (if found).

    - removeDocumentBySender(): this requires an address and calls the Corpus.removeDocumentBySender() method to remove all Document{} objects in corpus[] that were submitted by the provided address.
    - connectMetaMask(): this requires no additional parameters but does require the MetaMask browser extension be installed and enabled. This refreshes the browser session with MetaMask.
- HTML Corpus Interaction Administrative Frontend (corpus_server.js/corpus_index.html)
  - This is a basic HTML page built with Node.js that accepts inputs for the above Interaction Script methods and displays results and errors to the user. It is currently derived from previous submissions and will likely be cleaned up prior to the final project submission. However, because this particular frontend is intended for administrative users and not end users (there will be a second UI for the RAG/chatbot system that is intended for end users), the final submission will likely not be particularly flashy.

This project was deployed via Hardhat and Node.js in an Ubuntu system with NPM installed, using the commands located in the README.md file in the project root directory.