**Abstract**

Recent advancements in Retrieval-Augmented Generation (RAG) systems represent a significant development in natural language processing, combining the generative capabilities of language models with external information retrieval mechanisms to enhance factual accuracy and contextual relevance. They offer a particularly valuable opportunity to develop query tools for private and proprietary information or data that has not been included in a model's training corpus. We developed a proof of concept natural language query and response RAG (FolkRAG) for public data from the American Folklife Center (AFC) at Library of Congress (LoC) and developed a methodology to determine vectorstore parameters that optimize document retrieval accuracy for cultural heritage materials at scale.

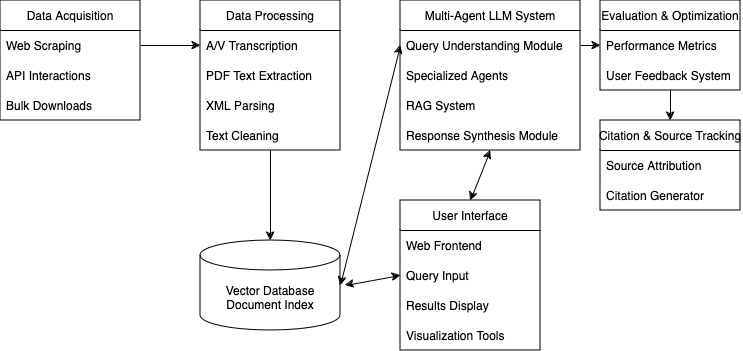
**I – Introduction**

A single archival collection can be, and often is, described and presented online in a variety of ways that incorporate different metadata and metadata schema housed in different locations that, while connected intellectually, may or may not be connected semantically. Elements of that collection are often displayed online in the form of digital surrogates that utilize additional technical and descriptive information that, again, may or may not link back to a central document that allows a researcher to make sense of the material.

The collections at Library of Congress (LoC), and specifically the digital representations of those of the American Folklife Center (AFC) upon which this experiment centers, adhere to this. A collection is typically represented by a catalog record in catalog.loc.gov which may or may not link to a finding aid in findingaids.loc.gov, if it has one; the finding aid may or may not link back to the catalog record; there may be digital versions of objects from the collection (image, text, audio, or video) in loc.gov/collections that are sometimes, but not always, linked from the catalog record or finding aid; those objects have metadata that mostly, but not uniformly, link back to the previously mentioned sources. Confused yet? Researchers (and indeed staff) often are.

INSERT DATA DIAGRAM

This is problematic for several reasons, but chief among them is the impossibility of searching all data sources at once. A researcher must rely on their own facility with archival research to successfully navigate a collection, and, when necessary, on the guidance of reference librarians who sometimes know these collections intimately, but often do not. Researchers and staff alike rely on these systems of description to guide them. Secondly, there is no way to interact with these discovery systems in a way that accounts for half-remembered details, threads upon which an individual can pull for a more natural entry point into a subject area - there is no way to ask questions without in-depth knowledge of how these systems work, which few truly possess. The aim of this project, therefore, is to build a system that addresses these concerns computationally without abandoning the core tenets of librarianship - service, accountability, context, and authority. In short, to allow natural language query and return natural language answers with citation via Retrieval Augmented Generation (RAG) and in doing so develop a methodology to determine vectorstore parameters that optimize document retrieval accuracy for cultural heritage materials.



*Figure II: System Architecture (probably should be reviewed)*

**II - Data Overview and Acquisition**

Note - All data for this project was acquired legally via public sources; access to internal LoC sources was neither requested nor required. All computing was performed on Amazon Web Services (AWS) g5.2xlarge Ubuntu Elastic Compute (EC2) instances provided by The George Washington University (GWU).

AFC, founded in 1976 when Congress passed the American Folklife Preservation Act, has fulfilled its charge to preserve and present folklife in all its diversity, and documents and shares the many expressions of human experience to inspire, revitalize, and perpetuate living cultural traditions. It houses the Archive of Folk Culture, which was originally founded as the Archive of American Folk Song in 1928. The holdings comprise documentation of traditional culture from around the world and feature multiformat materials in both analog and digital formats. Its digital collections generally contain photographs, audio recordings of oral history interviews or folksong, correspondence, and manuscript materials such as field notes or logs for recordings or photos, as well as archival collections. ADD CITATION: https://www.loc.gov/research-centers/american-folklife-center/about-this-research-center/

The catalog is typically the first place a public record of a library resource is created. It contains high level bibliographic information about an item, and allows that information, whether it pertains to a book, archival collection, or other format, to be located. The record is created by a cataloger using the Machine Readable Cataloging (MARC) format and is available as MARC XML at https://lccn.loc.gov/. A bibliographic item is typically represented by a single catalog record. ADD CITATION: https://dictionary.archivists.org/

Archival finding aids describe, in a broad sense, what a collection contains - if physical, its container list will document each box, folder, and sometimes item; if born-digital, its container list will document whatever arrangement has been imposed on the digital materials; if both, then a combination of each approach is used. Finding aids also provide contextual information about the record creators, related materials, provenance, and how the collection has been arranged, why, and by whom. They also utilize controlled vocabularies such as subject headings to provide researchers with a general impression of the topics covered in the collection. They take the form of structured XML known as Encoded Archival Description (EAD), and LoC's are located at https://findingaids.loc.gov/exist\_collections/ead3master/. An archival collection will typically be represented by a single EAD, written by an archivist once a collection has been processed and is available for research. ADD CITATION: https://dictionary.archivists.org/

Digital objects on loc.gov are arranged into collections that often, but not always, correspond to archival collections on findingaids.loc.gov or catalog.loc.gov. Objects are offered in a variety of audio, video, image, and document formats, and both JSON metadata and the objects themselves are publicly accessible and downloadable via the loc.gov application programming interface (API). Each digital object or file is typically represented by a single metadata record.

For our project, we wrote a web scraper to obtain individual EAD XML and MARC XML records for all AFC collections and utilized the loc.gov API to pull AFC digital collection JSON metadata at the file level and transform it into .csv. Since that metadata contains file locations and mime-types, we then structured and executed wget requests to download all AFC .txt, .pdf, .mp3, and .mp4 files. We extracted all .pdf text with tesseract optical character recognition (OCR) and transcribed (and, where appropriate, translated into English) all .mp3s with OpenAI's large Whisper speech recognition model (CITATION: https://github.com/openai/whisper). We scraped 158 EAD XML files, 158 MARC XML files, and obtained metadata for 48 digital collections. Our wget pulls followed by either OCR or Whisper transcription resulted in acquisition of 10,980 machine generated transcripts, 4,292 OCR files, and 29,778 library-created text files for a total of 45,050 documents and corresponding metadata records, representing the entirety of AFC's publicly available digital collections as of October 2024. Note that although .mp4 files were acquired, due to the computationally intensive nature of the Whisper large model and our limited timeframe they were ultimately not included in the project data. We also opted to exclude .wav and .mov files from this project.

**III - Metadata Processing**

Our loc.gov API queries resulted in two .csv files for each collection - search\_results.csv and file\_list.csv. Each request is based upon a search, and each search returns a list of resources and their descriptive metadata - this is returned as JSON and saved as a .csv. Each resource can contain any number of derivative files (for example, an image may be available as .jp2, .jpg, or .tif; a video may be available as .mp4 or .mov; audio may be available as .mp3 or .wav), and search\_results.csv contains links to the item pages where those files are available. It's this information that is used in a second API call to the item pages to generate file\_list.csv - the list of every file for every resource in the search, its type, and its location. Once those files were acquired, we developed a metadata processor to link each file with selected fields across a collection's file\_list.csv, search\_results.csv, EAD XML, and MARC XML before attaching that information to each chunk in our vectorstore.

The metadata processor takes a modular approach to integrating diverse archival description sources into a unified whole. It extracts and normalizes identifiers from URLs in various formats through pattern matching, generates potential matching patterns for different resource types, and implements thorough error handling and logging.

For EAD and MARC, the processor parses hierarchical elements like title, date, abstract, subject headings, and series titles, and preserves the contextual information these sources provide at the collection level. The system maps related fields across schemas while maintaining source attribution, addressing the key challenge of disconnection between finding aids, catalog records, and digital objects.

The main processing function first parses the search\_results.csv to create multiple mapping dictionaries that associate different forms of identifiers (base IDs, digital IDs, and resource URLs) with their corresponding metadata. It then processes the file\_list.csv to establish direct file-to-identifier mappings. For each file encountered, the processor attempts multiple matching strategies: first trying an exact match through file\_list mappings, then attempting matches through the various identifier dictionaries if the direct approach fails.

Each document's metadata is enhanced with file-specific information including its type (transcript, OCR, or text), original filename, and processing status. The unified metadata schema includes over 30 fields spanning administrative, descriptive, and technical metadata, ensuring that researchers can access collection context regardless of their entry point into the material.

The processor recognizes and handles AFC's unique identifier patterns, accommodating both legacy and current naming conventions. It also manages special cases like English translations of audio transcripts, matching them back to their source recordings through pattern recognition.

To ensure data integrity, the processor includes validation steps that verify each metadata field's presence and format. The system implements comprehensive logging that tracks successful and failed matches, providing detailed reports on the processing outcomes. This systematic approach to metadata processing and integration creates a foundation for reliable information retrieval and generation in the RAG pipeline, while maintaining archival and bibliographic context.

The final metadata schema incorporates fields for identification (original\_filename, file\_type, chunk\_id, total\_chunks, call\_number), descriptive information (title, date, created\_published, language, type), authorship and custody (contributors, creator, repository, collection, source\_collection), content description (description, notes, subjects, original\_format, online\_formats), rights management (rights, access\_restricted), geographic coverage (locations), resource location (url), as well as specialized fields from finding aids (collection\_title, collection\_date, collection\_abstract, series\_title) and catalog records (catalog\_title, catalog\_creator, catalog\_date, catalog\_description, catalog\_subjects, catalog\_notes, catalog\_language, catalog\_genre, catalog\_contributors, catalog\_repository, catalog\_collection\_id). Array fields like contributors, notes, and subjects are stored as JSON strings to preserve their multi-valued nature while maintaining compatibility with Deeplake's data structure. While there is some overlap between the kinds of information stored in some of these fields, it is preserved in the interest of cross-referencing and maintaining provenance.

**IV - Database Creation**

RAG systems represent a cutting-edge solution in natural language processing, combining the power of state-of-art large language models (LLMs) with targeted information retrieval. By supplementing LLMs with external data sources stored in a vector database, RAG systems can overcome some of the risks of hallucinations and the static nature of training data. The core components of RAG involve a retriever that locates and collects the necessary information from the vector database and a generator that incorporates this information to produce contextually rich and accurate responses. The retrieval component can operate using either dense or sparse vector spaces to ensure relevance and accuracy in the data retrieved, with the former capturing more complex semantic relationships between words and phrases that enable a more nuanced understanding and retrieval of information. The generator component then is an LLM that adapts its responses based on the information provided by the retriever.

The vectorstore utilizes DeepLake and LangChain as the underlying database technology, chosen for their ability to handle the scale and complexity of AFC's digital collections while maintaining efficient similarity search capabilities and metadata-based filtering. The system offers multiple embedding model options: HuggingFace's Instructor-XL, MiniLM, and Amazon's Titan embedding model. Each model presents different tradeoffs - Instructor-XL excels at understanding nuanced language and context but requires more computational resources, while Titan leverages AWS Bedrock's infrastructure for potentially faster processing at scale, although the high number of API calls required for a dataset of this size ultimately caused Titan embedding to take significantly longer than Instructor-XL. The choice between models, however, allows users to balance performance characteristics against their specific needs and infrastructure constraints. It also allowed us to test differences in retrieval accuracy between embedding models.

The vectorstore initialization process creates a comprehensive tensor structure that preserves both the textual content and its associated metadata. Each document is represented by a set of tensors including the raw text, embeddings, and all metadata fields established during processing (each field gets its own tensor). The system implements memory management through batch processing, with configurable batch sizes (defaulting to 100 documents) to handle the large volume of digital collection material efficiently and without overwhelming memory resources.

Document chunking is performed using LangChain's recursive character text splitter with configurable chunk size and overlap percentage (defaulted to 15% of the chunk size), preserving document boundaries and maintaining contextual coherence. For transcript files, the chunking process includes additional preprocessing to remove timecode annotations while preserving the temporal relationship between text segments. The configurable chunk size allows users to balance the granularity of text segments against the context window limitations of their chosen language models. It also allowed us to test how different chunk size selections impacted retrieval accuracy.

The embedding generation process incorporates several safeguards to ensure data integrity. A checkpointing system tracks progress and enables recovery from failures, crucial when processing tens of thousands of documents. The system maintains detailed logs of the embedding process, tracking successful operations and documenting any anomalies or failures at both the document and batch level.

The similarity search employs cosine similarity calculations to identify relevant documents, with index validation to ensure robust query results. The search function supports configurable top-k retrieval and includes metadata filtering capabilities. Each search result includes a similarity score and dataset index, enabling transparency in the retrieval process and facilitating result validation.

Error handling is implemented at multiple levels, from individual document processing to batch operations, with detailed logging that captures both successful operations and failures. The system includes memory optimization techniques, implementing garbage collection and Compute Unified Device Architecture (CUDA) memory cache clearing where appropriate to maintain stable performance during long processing runs.

This vectorstore implementation creates a foundation for reliable information retrieval while preserving the complex relationships and context inherent in archival materials. The result is a system that maintains the integrity of archival description while enabling natural language interaction with the collections.

**V - Approach**

Our methodology centered on evaluating different RAG configurations through a two-phase process. Initially, a small data sample was used to narrow down the most promising RAG configurations. We implemented both a basic naïve RAG retrieval and an advanced system incorporating a hypothetical document embedding (HyDE) mechanism ahead of the retriever, as HyDE strategies have shown promise at significantly improving document retrieval. Evaluation revealed two optimal chunking strategies and two embedding models for further analysis.

In the second phase, we employed the entire dataset to identify the optimal retrieval process. Three vectorstores were constructed using the approaches identified during the initial tests. The same RAG architectures were compared, in addition to a third advanced system. The HyDE mechanism was enhanced, and a re-ranking module was incorporated to reassess the relevance of the retrieved documents.

This approach not only enhances the accessibility of archival collections but also affirms the performances of retrieval strategies on vectorstores constructed using complex data structures. It empowers researchers to reliably engage with these systems using natural, intuitive queries, bridging the gaps between technical metadata, human inquiry, and comprehensive information.

Five primary vectorstores were retained for the final application, two following our testing on the subset of the data and three used for the entire dataset.

* Subset of the data embedded into chunks of 250 characters, encoded by hkunlp/instructor-xl.
* Subset of the data embedded into chunks of 1,000 characters, encoded by hkunlp/instructor-xl.
* All of the data embedded into chunks of 250 characters, encoded by hkunlp/instructor-xl.
* All of the data embedded into chunks of 1,000 characters, encoded by hkunlp/instructor-xl.
* All of data embedded into chunks of 250 characters, encoded by amazon.titan-embed-text-v2:0.
* All of data embedded into chunks of 1,000 characters, encoded by amazon.titan-embed-text-v2:0.

**VI - Architectures**

**Builds:** To develop an optimal RAG system, three architectures were developed and implemented, growing from a naïve RAG system to incorporating advanced methods, like hypothetical document embedding strategies and re-ranking. In all cases, the LLM used in this project is anthropic.claude-3-5-sonnet-20240620-v1:0. Depending on the user’s preference and for comparing purposes, two embedding models are primarily used throughout: hkunlp/instructor-xl (public) and amazon.titan-embed-text-v2:0 (private). A third embedding model is briefly used in early sample testing but soon after dropped as a consideration due to poor performance: all-MiniLM-L6-v2 (public).

* **Naïve RAG:** The naïve RAG architecture receives the query from the user, embeds it using the same model that was used to construct the vectorstore (titan or instructor), and sends the top\_k documents relevant to the query to the LLM for response generation.

A diagram of a software company

Description automatically generated

* **RAG with Hypothetical Document Embeddings (HyDE):** A more advanced RAG architecture includes a HyDE component that aims to provide the retriever with additional context to improve retrieval accuracy. The user’s query is supplied to an LLM that generates a hypothetical response. The query and response is combined to form a hypothetical document that gets embedded and searched in the vectorstore. The top\_k matching documents are then provided back to the LLM for response generation.

A close-up of a computer screen

Description automatically generated

* **RAG with Advanced Hypothetical Document Embeddings and Re-ranking (HyDER):** The final architecture increases the complexity of the HyDE component and adds a re-ranking feature at the end to prioritize the best scoring documents and reduce noise.The hypothetical document generator from the previous HyDE architecture is expanded to produce two additional hypothetical documents. The LLM receives the user’s query and generates two derivative queries and responses to all three for three hypothetical documents. Each are embedded and passed to the vectorstore that retrieves top\_k matching documents for *each* hypothetical document. All of the results are combined and de-duplicated and then passed to a re-ranking mechanism. The pooled documents are re-ordered according to their final relevancy scores and the best scoring top\_k documents are retained and sent to the generator to produce a response for the user.

A screenshot of a computer

Description automatically generated

**Hypothetical Document Embeddings Methodologies:** The hypothetical document generator is integrated early in the system using an LLM to generate and process variations of the users' query and their responses.The LLM generates an initial response for the original query, rewrites the query into slightly different variations, and generates answers for each variation. For the HyDER architecture, each of the three query-and-responses are generated using progressively higher temperatures (i.e., 0.7, 0.8, 0.9) to preserve factual integrity but also ensuring a degree of variety; the HyDE system applies a 0.7 temperature. All of the generated content is passed through a cleaner to compensate for instances when the LLM does not generate adequate output, or if the output includes language not relevant to the objective. For example in some cases, the LLM refrains from forming a response out of its perceived copyright concerns. In such instances, valid output is taken from the other hypothetical documents to assure each hypothetical document is complete for passing forward.

* Derivative query generation prompt:

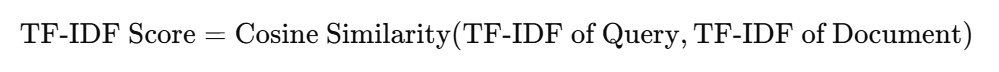
*Rewrite this query to be slightly different but similar in meaning: {query}*

* Hypothetical response generation prompt:

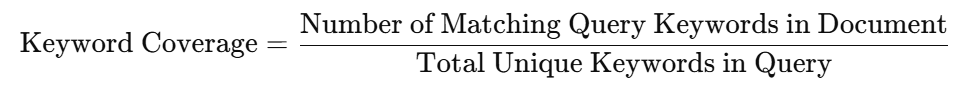
*You are a document that answers this question {query}.*

*Write a short, natural paragraph that directly answers this question. Include additional relevant information if possible.*

**Re-Ranker Details:** The re-ranking function employs a multi-faceted approach to assess and rank documents based on their relevance to a query. The process integrates *sklearns* TF-IDF scores, a basic keyword coverage calculation, and document freshness based on its date to achieve an effective ranking mechanism. The TF-IDF portion assesses the relevance of documents’ content to a query. Vectorization tokenizes and transforms the text of all documents, as well as the query, into a matrix of TF-IDF features. Words that are frequent in a single document but rare across the corpus are given higher scores. Cosine similarity is calculated between the query's TF-IDF vector and each document's TF-IDF vector to get a relevance score for each document. A higher score for a document indicates a greater alignment with the query's content.



To further refine relevance, the function calculates keyword coverage by determining the proportion of unique words in the query that are explicitly present in each document's text. For each document, the number of unique query words found in the document is divided by the total number of unique words in the query. This calculation results in a keyword coverage score, which rewards documents that include a higher proportion of the query terms, emphasizing direct textual matches.

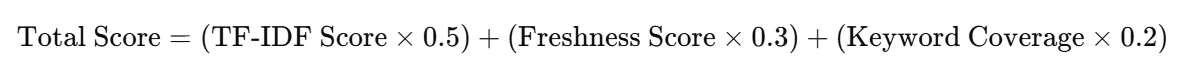


To prioritize recent documents, a freshness score is computed based on the age of the document relative to the current date. Each document’s metadata is checked for a timestamp and, if available, assigned a score based on its proximity to the present day, with newer documents receiving a higher score.

A black text on a white background

Description automatically generated

The final relevancy score integrates the TF-IDF and similarity calculations with the recency and keyword coverages scores to determine which documents are most related to the input query. Different weights are assigned to each to emphasize TF-IDF and cosine similarity. Based on the total score, documents are re-ordered and the best scoring top\_k documents are retained for use in response generation.



**Generator:** The top\_k retrieved documents each contain text that gets combined into a single ‘context’ variable. For token length efficiency and to avoid mixing context with unmatching metadata, only the best scoring document’s metadata is used in generating a response to the user’s original query.

*Human: Please answer the following query based on the provided context and metadata.*

*Query: What are the key findings from the 2024 climate study?*

*Context: [Relevant text from top-ranked documents]*

*Metadata: [Relevant metadata from top-ranked document]*

*Instructions:*

*1. Answer the question using ONLY the information provided in the Context and Metadata above.*

*2. Do NOT include any information that is not explicitly stated in the Context or Metadata.*

*3. Begin your answer with a direct response to the question asked.*

*4. Include relevant details from the Context and Metadata to support your answer.*

*5. Pay special attention to the recording date, contributors, and locations provided in the metadata.*

*6. Inform the user of what document filename they can find the information in.*

*Your Answer here:*

**VII - Evaluation**

**Phase I – Narrowing retrieval strategies:** A small sample of data was extracted and used for initial testing. Vectorstores built in chunks of 250, 500, 1,000, and 2,000 were examined using the three different models. Naïve and HyDE architectures were also compared to gauge the efficacy of HyDE as a RAG strategy. Each system configuration was tested with 100 questions, each accompanied by the unique filename of the document that contained the question’s correct answer. Accuracy was measured on whether the correct document was included in the top\_k filenames returned by the retriever.

As expected, all-MiniLM-L6-v2 as an embedder and the Naïve architecture proved inadequate. The basic HyDE architecture showed significant improvements in retrieval accuracy, generally increasing it between ten and twenty percentage points in most system configurations. Surprisingly, the vectorstore constructed using embedding chunk sizes of 250 characters, a particularly short length of text, competed with that of the same using 1,000 sized chunks for the optimal vectorstore. The most promising system architectures consisted of the basic HyDE generator, and instructor-xl or titan as the embedding model to construct the vectorstores in chunk sizes of 250 or 1,000. In testing on the sample set, 90% retrieval accuracy on the 100 sample test questions was achieved by within a top\_k of four documents.

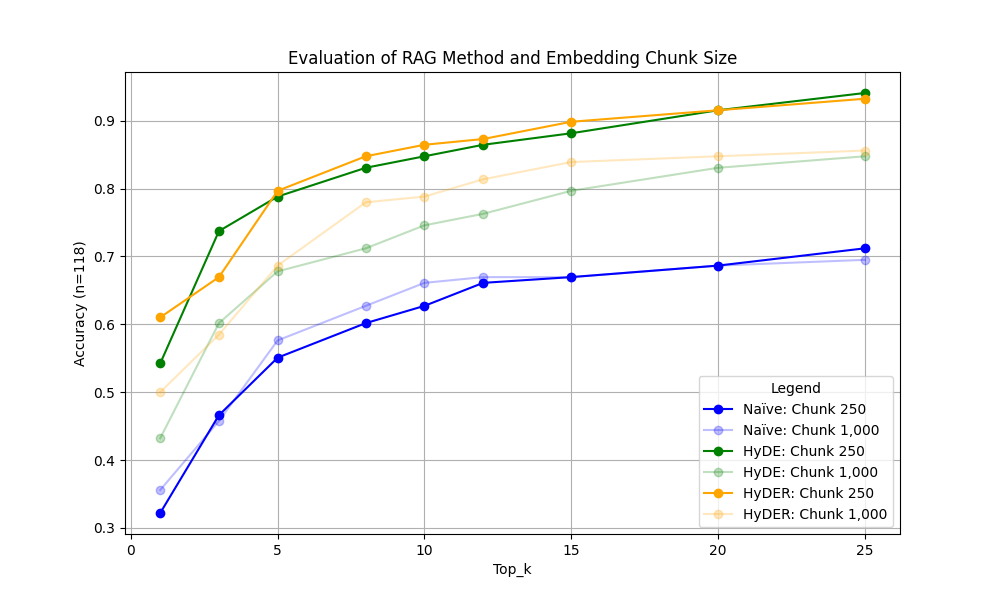
A group of graphs with different colored lines

Description automatically generated

**Phase II – Applying to all data:** Evaluation of the RAG system’s configurations was expanded to the entire dataset. Vectorstores based on 250 and 1,000 chunking strategies consisted of 1.6 million and 400,000 million chunked documents, respectively. Similar to the sample data, 118 new questions were used to gauge retrieval accuracy. The same naïve and HyDE architectures were again compared, as well as HyDER now being included.

Although both instructor-xl or titan embedding models showed promising results, sample accuracies were compared between the two to identify which provided better retrieval results. For efficiency and considering computation and time constraints, only the better performing one was applied in all system configurations. Under the HyDER architecture, separate tests with top\_k values of one and three on vectorstores chunked by 250-character blocks showed noticeably different retrieval accuracies. The retriever mechanism with titan-generated embeddings showed accuracies of 45% and 59% for top\_k values of one and three, respectively, while the instructor-generated embeddings showed accuracies of 61% and 67% for the same top\_k values. Therefore, titan was dropped and all RAG systems and components were compared using the instructor-xl model as the embedder.

Final evaluations showed close performances between the HyDE and HyDER architectures, with HyDER narrowly performing better. Additionally, similar to our findings on the subset of the data, the vectorstore constructed using 250-character sized chunks provided better results. Both the architectures on 250 chunk-sized vectorstores achieve 90% accuracy at or just beyond a top\_k value of 15, and plateau at around 94% after top\_k of 25.



**Final Insights:** The close performances between the HyDE and HyDER systems suggest that the significantly more complex HyDER may not be necessary to achieve optimal results. Basic HyDE mechanisms in most instances are likely sufficient to drastically improve retrieval accuracies.