Lil Lisa v2 Support Chatbot

Carlos Escobar

Introduction

The idea of talking to machines has been an aspiration of many for decades. Until the advancement of transformers, a key technology in large language models (LLMs), reliably talking to machines seemed impossible. With the development in LLMs such as ChatGPT, we can now seamlessly chat with these machines, giving rise to numerous applications in code generation, question answering, and much more. These models need to be robust, being able to handle complex requests and properly executing the sequence of actions it has planned out. One critical aspect in enhancing the functionality of conversational agents lies in their ability to access and integrate information from external sources. While LLMs are trained on a vast corpora of text, encompassing extensive knowledge from the internet, they may still encounter situations where specific information is lacking or outdated.

To address this limitation with Lil Lisa v2, I created a Retrieval-Augmented Generation (RAG) system with the capability to query a database, enhancing answers to a given query from a user. In this study, I explored several chunking and indexing methods, aiming to evaluate its impact on response quality and efficiency.

Background

Vector Search

Vector search is a fairly new concept dating back to the early 2000s, and there have been many advancements since. Vector search is used to compute the similarity between words and phrases by comparing the difference of vectors in a space, usually by calculating Euclidean distance from one point to another which modern computers can do extremely fast. In 2018, Bert created the first word embedding model utilizing neural networks spanning 768 dimensions, and in present-day, OpenAI uses 3072 dimensions, hoping to maximize semantic understanding of a phrase, both being examples of bi-encoders. Vector search tends to work better on phrases or sentences rather than single words, which are better handled by keyword search algorithms like TF-IDF (Term Frequency-Inverse Document Frequency) or

BM25 (Best Match 25) which relate word frequencies. For example, when you search for ”Nike”, with the vector search you would get results for ”Nike”, ”Jordan”, ”Adidas” since they are all semantically similar while a keyword search would retrieve information particularly about ”Nike”.

However, a challenge arises in encoding these phrases in a consistent and valid way so that the difference between phrases makes sense to us intuitively. Additionally, more dimensions doesn’t mean better performance; there must be a balance between it and the complexity of the data. Too many dimensions can lead the model being more prone to noise, whereas too few dimensions may not fully capture the nuances of the data.

Keyword Search

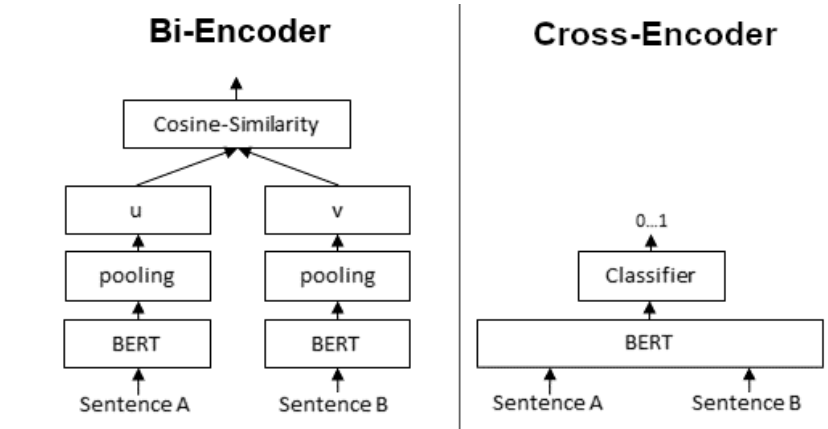
Keyword search is a method, first seen in the 1990s as WebCrawler, that relies on matching exact words in a query with those in a document to compute relevancy. Algorithms like TF-IDF and BM25 excel in retrieving precise information, making it ideal for specific queries, but struggling with complex ones as it does not take into account the semantic meaning of a query. Additionally, keyword search is significantly slower than vector search.

Bi-Encoders vs. Cross-Encoders

Cross-Encoders work similar to Bi-Encoders as they both calculate relevance between two pieces of text, but they work differently behind the scenes. Bi-Encoders take in two text chunks which are embedded independently; on the other hand, cross-encoders take in both text chunks together and are jointly encoded, capturing a deeper understanding of their semantic similarity.

Why do Bi-Encoders exist then? Well, cross-encoders frankly do not scale well for large amounts of documentation. Instead, we could use a mix of both by retrieving the top-50 documents embedded by a bi-encoder then rerank these using a cross-encoder.

[FOUND ONLINE] *Diagram of underlying relevancy computation*



Experiments

Chunking Comparison - Lexus

I will use three different chunking methods (hierarchical, semantic, and sentence) with three different embedding models (OpenAI's text-embedding-3-large, BAAI's bge-large-en-v1.5, Sentence Transformers' all-MiniLM-L12-v2) each. They have 3072, 1024, and 384 dimensions respectively. There are 9 variations in total.

What do these chunking methods mean?

I will be conducting some testing to find the best variation using a single document (article on a Lexus RC F)

For each variation, this will be conducted:

Preprocessing

- Using LlamaIndex, chunk the document into smaller pieces and store them in nodes, which contain crucial information such as the ID
- The text associated with each node is embedded and stored in a LanceDB database

Testing

- Using DeepEval, for each node, generate a few complex questions only that node's text could possibly answer
- Record the ID tied to the node
- Embed each query and using vector search with Euclidean distance, return the vectors in the database from closest to farthest from the embedded query
- Associate the vector back to its node and record the rank of where it appears in the list
- Keep in mind, the optimal place to find the node is at the top of the list
- Node Count:
    - Hierarchical: 39 (128, 512, 2048 token text size hierarchy)
    - Sentence: 16 (chunked using 256-token splitting and 20-token overlap)
    - Semantic: 7

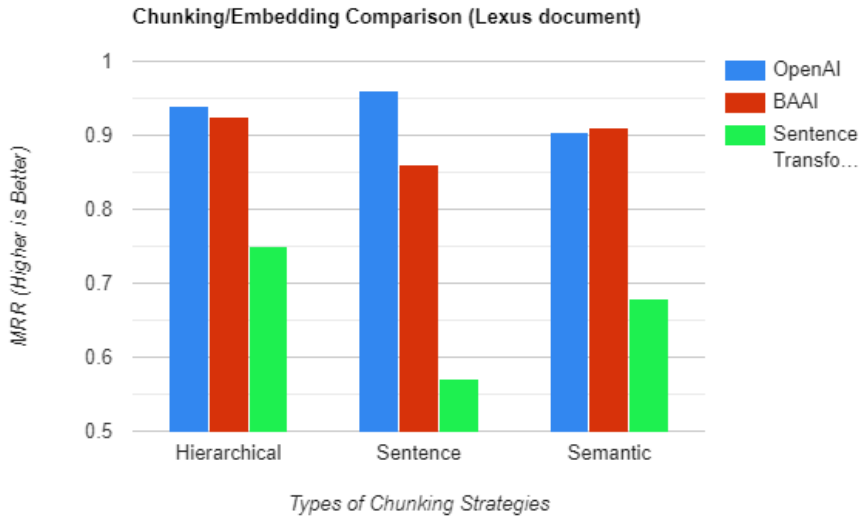
- Question Count used for Testing:
    - 75

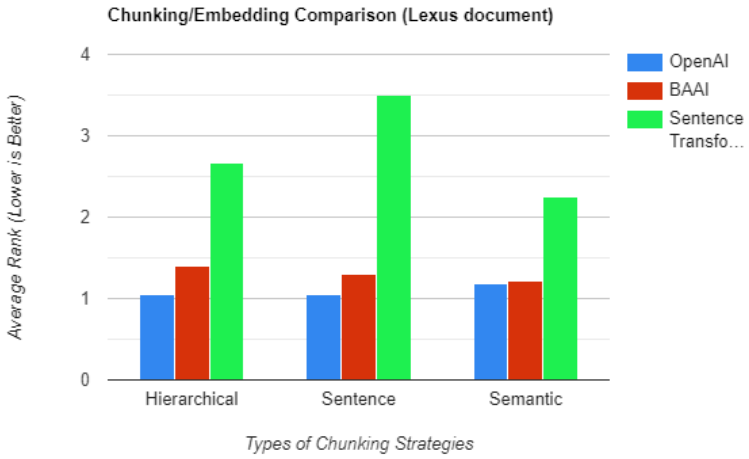
Metrics returned:

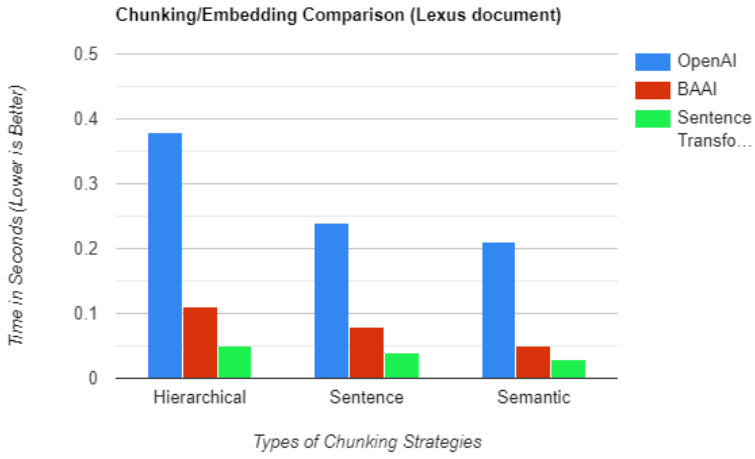
- Time to retrieve all nodes for each variation, per question
- Average Rank
- MRR (Mean Reciprocal Rank)

[What do these metrics mean?]

Results







Due to the high amount of preprocessing needed for semantic chunking and its not so convincing results, I will not be continuing with this approach on future testing with larger documents.

After this, the top 3 variations seem to be:

- OpenAI, Hierarchical
- BAAI, Hierarchical
- OpenAI, Sentence

Searching/Embedding Comparison - SQuAD v2

I will be downloading a dataset from huggingface.co and using it to test my retrieval algorithms. Due to the nature of the dataset having preset queries and corpus', there is no need to chunk them down, but instead store the corpus' directly into the database. Additionally, the dataset has 11873 queries, but we will only be testing on the first 250 queries for the sake of time. I will be doing this using the same OpenAI and BAAI embeddings. Instead of using solely vector search when retrieving nodes from the database, we will be using these five approaches:

1) Vector Search
2) Full Text Search (FTS), using BM25
3) Hybrid Search (70% Vector Search score, 30% FTS score)
4) Best approach of the above 3, then reranking the top-40 nodes using a cross-encoder (ms-marco-MiniLM-L-12-v2)
5) Step 4, then reranking the top 5 documents using Colbert v2.0 as a final way to get the most relevant document to the very top (helps with excluded middle when synthesizing)

Node Count:

- 1204

Question Count used for Testing:

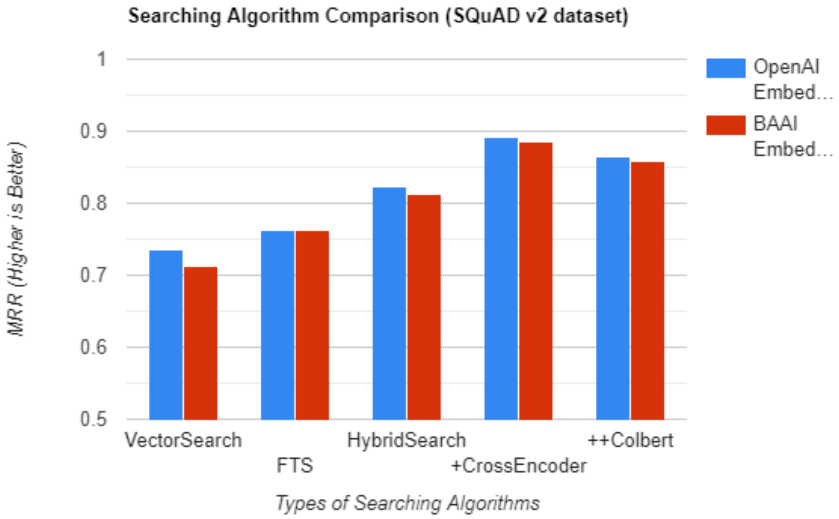
- 250

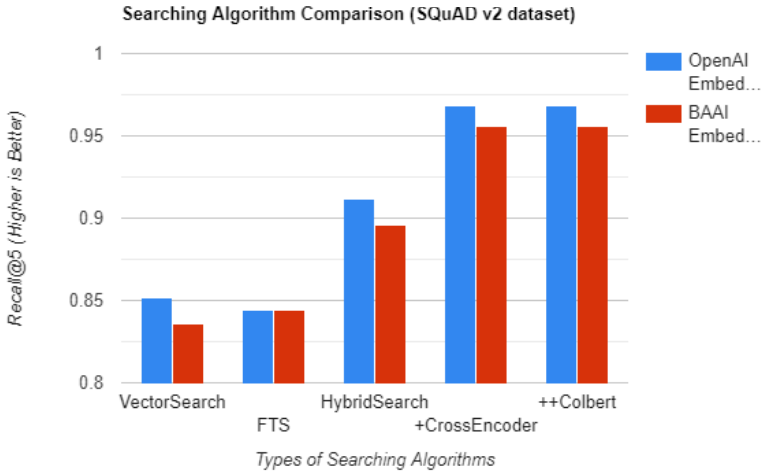
Metrics returned:

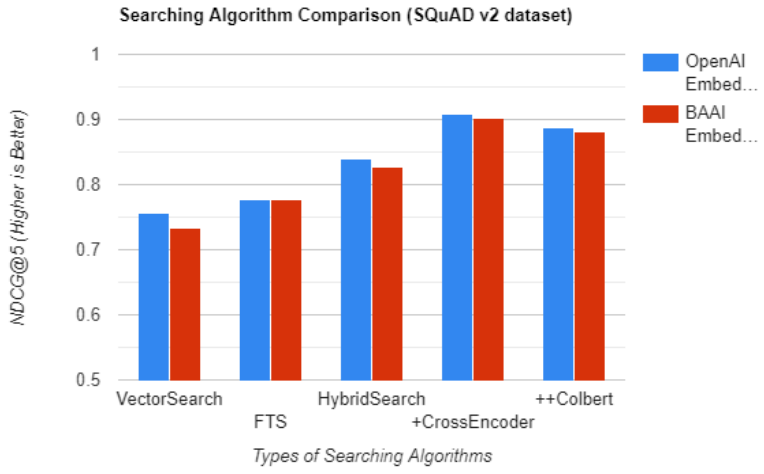
- Time to retrieve top-50 nodes for each variation, per question
- Average Rank
- MRR (Mean Reciprocal Rank)
- Recall@5

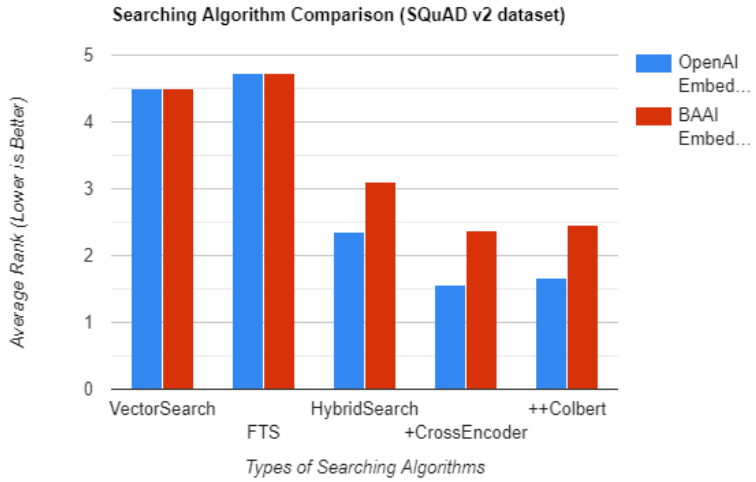
- NDCG@5 (Normalized Discounted Cumulative Gain)

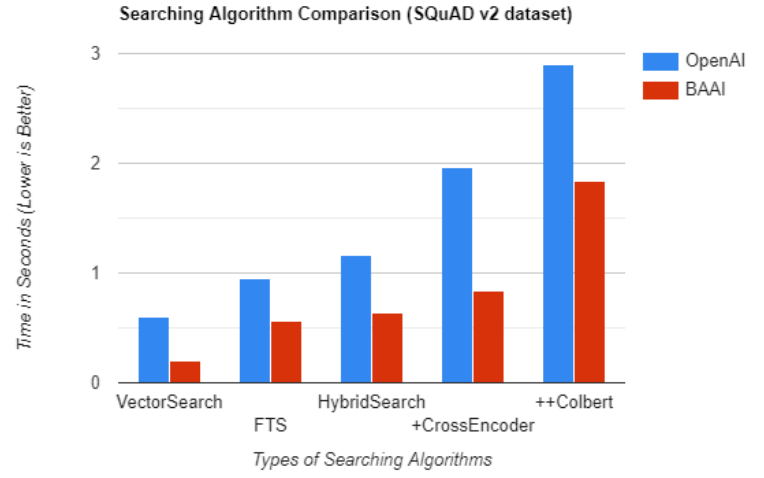
Results











One could see a time tradeoff in exchange for better retrieval with OpenAI embeddings, but 2 seconds per query is not a lot of time in order to maximize retrieval quality in our RAG system.

Chunking/Indexing Comparison - Radiant Logic

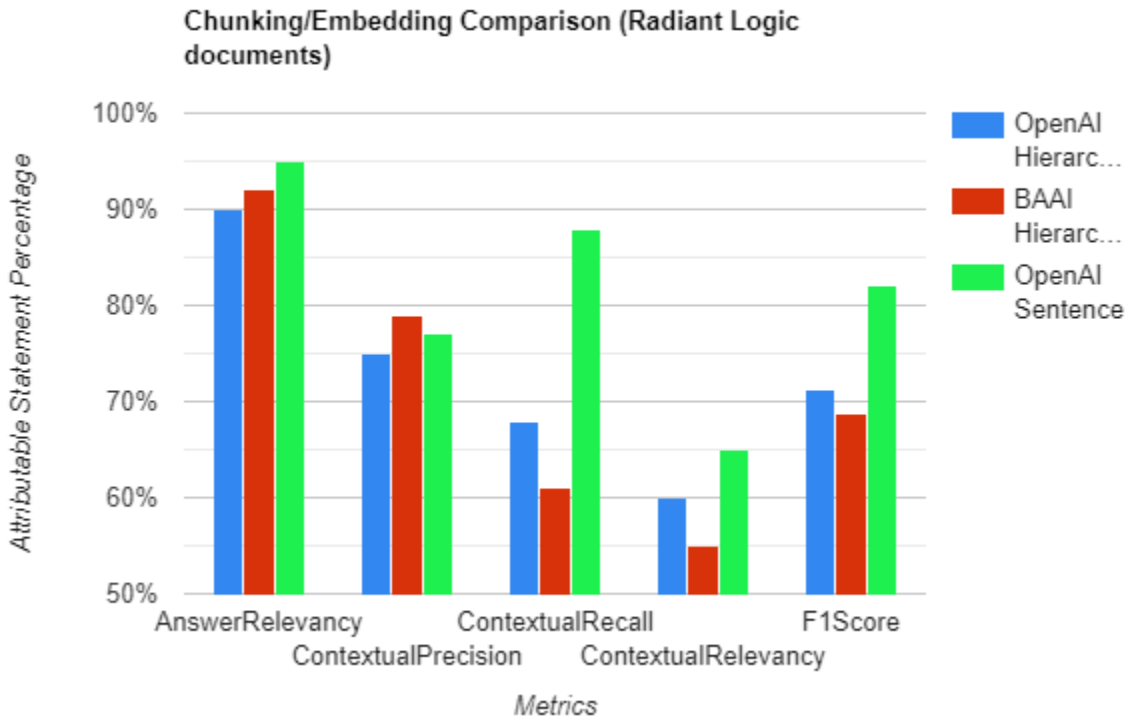
Using DeepEval, I generated 40 complex questions and “ground truths” that my RAG system could answer and try to resemble. I tested 3 variations (OpenAI - Hierarchical, BAAI - Hierarchical, OpenAI - Sentence) by running it through my retrieval system of Hybrid+CrossEncoder+Colbert, extracting the 2 most similar nodes, then sending a request to the GPT-4 API to synthesize a response. Due to the nature of the different chunking methods, initially, 4 nodes are returned by the sentence-splitting database, and 20 leaf nodes are returned by hierarchical chunking, in which they are merged into an unknown amount of nodes. They are then reranked and the top 2 are used in answer synthesizing.

Metrics returned:

- Answer Relevancy
  The answer relevancy metric measures the quality of your RAG pipeline's generator by evaluating how relevant the actual_output of your LLM application is compared to the provided input. deepeval's answer relevancy metric is a self-explaining LLM-Eval, meaning it outputs a reason for its metric score.
- Contextual Precision
  The contextual precision metric measures your RAG pipeline's retriever by evaluating whether nodes in your retrieval_context that are relevant to the given input are ranked higher than irrelevant ones. deepeval's contextual precision metric is a self-explaining LLM-Eval, meaning it outputs a reason for its metric score.
- Contextual Recall
  The contextual recall metric measures the quality of your RAG pipeline's retriever by evaluating the extent of which the retrieval_context aligns with the expected_output. DeepEval’s contextual recall metric is a self-explaining LLM-Eval, meaning it outputs a reason for its metric score.

- Contextual Relevancy
  The contextual relevancy metric measures the quality of your RAG pipeline's retriever by evaluating the overall relevance of the information presented in your retrieval_context for a given input. DeepEval's contextual relevancy metric is a self-explaining LLM-Eval, meaning it outputs a reason for its metric score.
- F1 Score
  Two times the product of the Contextual Precision Score and the Contextual Recall Score, divided by the sum of the Contextual Precision Score and the Contextual Recall Score

Results



Based on the F1 score, the two best chunking/indexing strategies are OpenAI Sentence Splitting and OpenAI Hierarchical Splitting, barely edging out its BAAI counterpart.

Chunking/Indexing Comparison - Radiant Logic pt.2

A clear winner in the reranking strategy was not present. For each chunking variation, I will be testing these four methods:

1) Hybrid Search (70% Vector Search Score + 30% Full-Text-Search Score)
2) Hybrid Search + Reranking top-k nodes using a cross-encoder (ms-marco-MiniLM-L-12-v2)
3) Hybrid Search + Reranking top-k nodes using a late interaction model (colBERT v2.0)
4) Step 2, then reranking the top-5 nodes using colBERT v2.0

k=30 for Chunking Variation 1, k=40 for Chunking Variation 2

A little twist to the chunking method will be applied and explained below.

Preprocessing

- Using a tool, found on LlamaHub, specialized in reading in Markdown files and splitting them by heading/subheading, chunk the files into structured nodes.
    - Chunking Variation 1: Store them as is
    - Chunking Variation 2: Since users mostly ask short, specific questions, create 128-token "child nodes" from these chunks as my idea is that the embeddings related to a child node contain very specific information about a certain topic that when hybrid search is conducted, it should pull the "correct" node rather easily. Store these in the database.
- Before ingesting into a LanceDB database, extract metadata such as the "Title", "Description", and "Version" of a file and tag this along with the respective node.
- The text associated with each node, along with parts of the metadata, is embedded and stored in a LanceDB database

Testing

- Using a set of 37 golden Question-Answer (QA) pairs generated from Lil Lisa v1, run through each variation and record the rank of the node that has sufficient information to answer the question

Node Count:

- Chunking Variation 1: 14060
- Chunking Variation 2: 37380

Question Count used for Testing:

- 37

Metrics returned:

- Time to retrieve top-k nodes and rerank accordingly, per query
- Average Rank
- MRR
- Recall@5
- NDCG@5

Results

Table 1: Performance Comparison of Chunking/Indexing Methods

Method	Average Rank	MRR	Recall@5	Recall@10	Recall@20	NDCG@5	NDCG@10	NDCG@20	Time Per Query (s)
Chunking Variation 1 + Hybrid Search	3.865	0.622	0.838	0.865	1.000	0.668	0.677	0.710	**1.020**
Chunking Variation 1 + Hybrid Search + Cross-Encoder	**1.703**	**0.858**	**0.973**	0.973	1.000	**0.885**	**0.885**	**0.892**	**2.034**
Chunking Variation 1 + Hybrid Search + ColBERT v2.0	**1.892**	0.765	0.946	**1.000**	1.000	0.805	0.823	0.823	10.293
Chunking Variation 1 + Hybrid Search + Cross-Encoder + ColBERT v2.0	**1.865**	0.786	**0.973**	0.973	0.973	0.832	0.832	0.839	4.018
Chunking Variation 2 + Hybrid Search	4.162	0.570	0.865	0.892	0.946	0.637	0.646	0.661	**1.200**
Chunking Variation 2 + Hybrid Search + Cross-Encoder	1.946	**0.858**	0.946	0.946	1.000	**0.877**	**0.877**	**0.891**	2.466
Chunking Variation 2 + Hybrid Search + ColBERT v2.0	2.378	0.793	0.919	0.973	0.973	0.818	0.836	0.836	13.738
Chunking Variation 2 + Hybrid Search + Cross-Encoder + ColBERT v2.0	2.135	**0.814**	0.946	0.946	1.000	**0.844**	**0.844**	**0.858**	4.61

Entries highlighted in **green** indicate the best performance, entries highlighted in **blue** indicate the second best performance and entries highlighted in **red** indicate the third best performance.

<u>Chunking/Indexing Comparison - Radiant Logic pt.3</u>

I will be testing the top 3 variations from the previous experiment:

- Chunking Variation 1 + Hybrid Search + Cross-Encoder
- Chunking Variation 2 + Hybrid Search + Cross-Encoder
- Chunking Variation 2 + Hybrid Search + Cross-Encoder + ColBERT v2.0

<u>Preprocessing</u>

- Using a set of 23 golden QA pairs generated from Lil Lisa v1 and an additional 50 QA pairs generated using Ragas, run through each

variation, and send the top-5 nodes along with a prompt to synthesize a response

Testing/Metrics:

- Answer Correctness
  Gather the query, ground-truth, and generated answer and send a request to the GPT-4 API asking it to assign a score of "1" if the generated answer is correct and a "0" otherwise
- Contextual Precision
- Contextual Recall
- F1 Score
- Time (s)
  How long, on average, to answer each query.

Node Count:

- Chunking Variation 1: 14060
- Chunking Variation 2: 37380

Question Count used for Testing:

- 73

Results

Table 2: Performance Comparison of Chunking/Indexing Methods

Method	Answer Correctness	Contextual Precision	Contextual Recall	F1 Score	Time Per Query (s)
Chunking Variation 1 + Hybrid Search + Cross-Encoder	**0.877**	0.971	**0.944**	**0.948**	**3.19**
Chunking Variation 2 + Hybrid Search + Cross-Encoder	0.822	**0.983**	0.899	0.905	3.440
Chunking Variation 2 + Hybrid Search + Cross-Encoder + ColBERT v2.0	0.849	0.961	0.918	0.920	4.468

Entries highlighted in green indicate the best performance.

Conclusion

In both of the previous tests, the combination of 'Chunking Variation 1 + Hybrid Search + Cross-Encoder' had the best overall performance.

My initial idea behind the hierarchical nodes didn't seem to work as expected as OpenAI's text-embedding-3-large's 3072 dimension vectors were enough

to retain the semantic meaning of large chunks created by Chunking Variation 1. I experimented with the use of Corrective RAG, but results were omitted for clarity as they did not perform well. Similarly, for Chunking Variation 2, I assessed whether to rerank the nodes whenever they were child nodes or after they were merged, and reranking as child nodes seemed to be slightly better, and that is what you see in the result tables above.

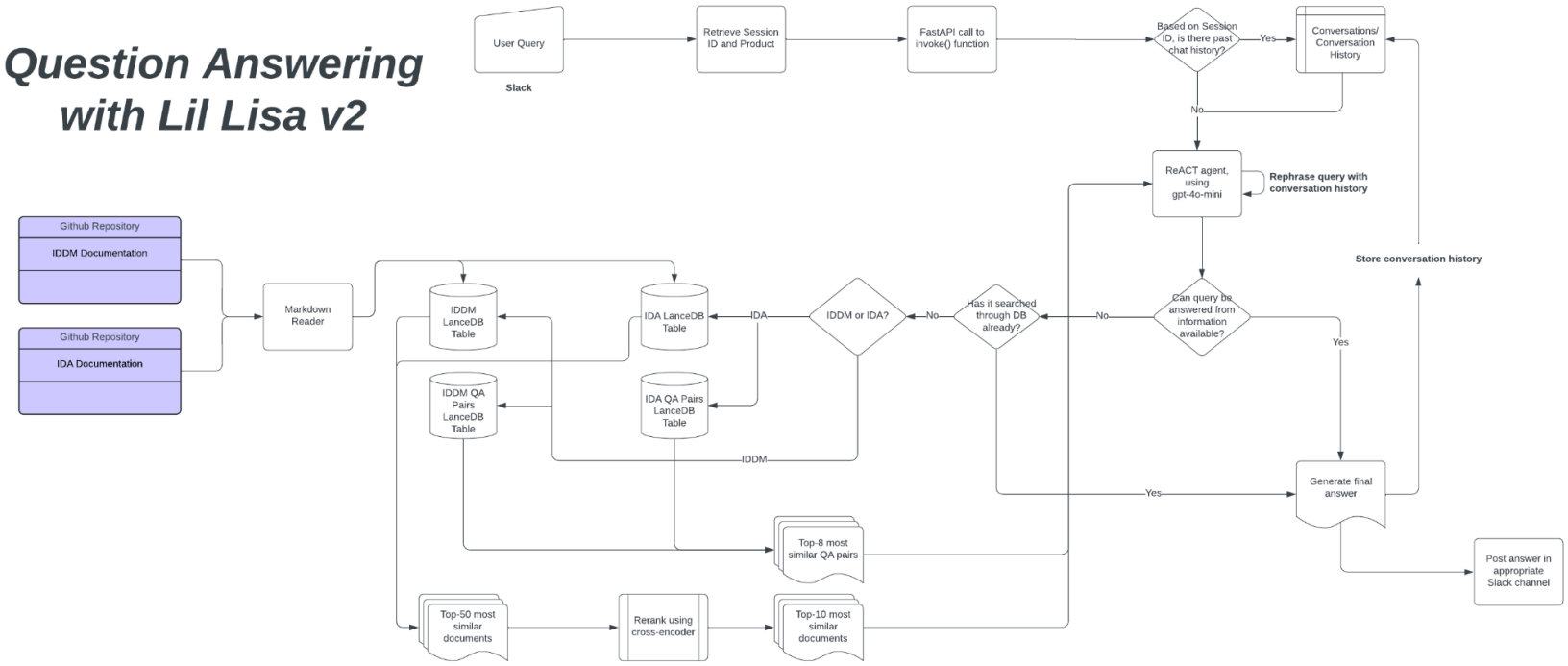
Using a cross-encoder as the lone reranking strategy is the way to go. It is fast and gains a semantic understanding deeper than using a bi-encoder. On the other hand, Colbert v2.0 brought relevant items down and was slow.

Improvements over Lil Lisa v1

- No need to tag the bot when you want an answer to your question (unless 2+ people in conversation)
- Multi-turn conversation
- If relevant context doesn't exist in the documentation, the agent is capable of telling a user the question is out of scope
- An "Expert" is invited into the conversation if user gives an "SOS" reaction
- Bot can be integrated outside of Slack, such as web portals
- We now have Lil Elvis! (for IDA queries)
- "/" commands have been streamlined making it easy for "expert" to review and update QA pairs accordingly
    - "/get_conversations" command has 3 options:
        - Option 1: "user". Gets all of the conversations endorsed by non-experts
        - Option 2: "expert". Gets all of the conversations endorsed by experts
        - Option 3: "unendorsed". Gets all of the conversations that were not endorsed
- Knowledge base is connected to active github repositories, allowing the "/rebuild_docs" method to access up-to-date information

- The answer provides relevant documentation links used to generate the answer
- Version-specific answers
- Bot deterministically returns an expert answer if it is found

Architecture/Diagram



Preprocessing

- The first step in the process is to extract versions, if any
- Filter QA Pairs and Documentation based on the versions extracted
- If a query is almost an exact match to an existing QA pair, this will bypass searching through the documentation and return the QA pair instead

Postprocessing

- If a query is relevant but not an exact match to existing QA pairs, it will be prepended to the LLM generated answer
- If a query is only somewhat relevant, it will be appended to the LLM generated answer
- The answer provides relevant documentation links used to generate the answer

Future Enhancements

- Allow users to provide screenshots with their questions
- Return screenshots/images with answers
- Response streaming
- Citations

- Remove documentation links from answer if query is out of scope

Appendix

LlamaIndex overview: https://docs.llamaindex.ai/en/stable/
LanceDB start guide: https://lancedb.github.io/lancedb/basic/
DeepEval start guide: https://docs.confident-ai.com/docs/getting-started
Ragas overview: https://docs.ragas.io/en/latest/index.html
BERT Paper: https://arxiv.org/abs/1810.04805
ReACT Paper: https://arxiv.org/abs/2210.03629
Link to the Github repository: https://github.com/Carlos-A-Escobar/rocknbot