Building, Evaluating, and Deploying Retrieval-Augmented Generation Applications

Nick Bukovec

A senior project presented for the degree of
B.S. Computer Science

Noyce School of Applied Computing
California Polytechnic State University
United States of America
June 10, 2024

1 Introduction

In the past two years, large language models (LLMs) have become prominent tools for answering questions across a large number of domains. Publicly available LLM applications, such as ChatGPT, now have millions of daily users using their text-generation services (Hu, 2023). However, despite how often they are used, LLMs do not always generate factually correct answers to users' questions. For example, Bhattacharyya et al., 2023 found that ChatGPT fabricated 47% of references when generating short medical papers. The goal of this project is to build a full application which uses retrieval-augmented generation (RAG) to factually answer questions. We aim to cover each step of the process, including extracting and loading the data into a retrieval model, developing and evaluating different LLM pipelines, deploying these pipelines with a REST API, and creating a web application for users to interact with our pipelines.

2 Background

According to Zhao et al., 2023, large language models like GPT-3 are pre-trained on vast datasets and use a transformer architecture to understand and generate human-like text. These models are fine-tuned for specific tasks, allowing them to perform well in applications such as question answering and translation. When answering questions, LLMs utilize a process called "in-context learning," where they generate responses based on the context provided by the user query. This involves processing the input text through multiple layers of attention mechanisms that help the model focus on relevant parts of the input, generating contextually appropriate answers. For complex reasoning tasks, strategies like chain-of-thought prompting, which involves generating intermediate reasoning steps, further enhance the model's ability to generate accurate answers.

Retrieval-augmented generation is a technique for eliciting better responses from an LLM by combining it with a retrieval model (Lewis et al., 2020). A retrieval model works by encoding documents into latent vectors that aim to represent the semantic meanings of the documents. When a query is passed into the model, retrieval models encode the query into a latent vector and then find the top-k most similar documents by calculating a similarity score against their corresponding latent vectors. These documents are then passed on to the language model along with the query to effectively and factually answer the query. When compared with fine-tuning, another popular method for knowledge injection, RAG systems outperform in accuracy when generating answers to knowledge-intensive questions (Ovadia et al., 2024).

3 Previous Work

Our application uses and builds off some of the latest techniques and tools in information retrieval and natural language processing. Santhanam et al., 2022 presents ColBERTv2, a retrieval model which encodes each document into

multiple vectors rather than a single vector embedding. The model uses a late interaction architecture to encode the query and compare its embedding matrix to the matrices of the documents to get the top-k more relevant documents. We use ColBERTv2 as our retrieval model to fetch relevant Wikipedia abstracts to answer our queries.

We use the dspy library to build out our systems, which uses the demonstrate-search-predict (DSP) framework proposed in Khattab et al., 2023. Systems that implement the DSP framework use training examples to demonstrate to the language model the desired behavior in each step of generation. They also implement a retrieval model to retrieve relevant documents which contain facts to later be used in the predict step. We implement our pipelines with the dspy framework due to its simple yet elegant way of defining complex LLM programs. However, we only leverage demonstration for the multi-hop RAG pipeline.

We base our multi-hop RAG pipeline off of Khattab et al., 2021 and their Baleen system. Baleen answers complex questions through a loop of fetching relevant documents from a retrieval model, condensing the passage into relevant facts, and then generating a new query for the next pass. Baleen outperforms existing systems when evaluated against a many-hop answer verification dataset. We modify Baleen in our application by replacing the proposed FLIPR retrieval model with ColBERTv2. We also reduce the two-step fact condenser into a single LLM call.

4 System Design

4.1 Data Pipeline

We use Wikipedia as our data source for retrieval. To do so, we create a pipeline to extract the first paragraph from every Wikipedia article as of April 1st 2024. We start with a Wikimedia enterprise HTML dump, which contains the HTML contents of every Wikipedia article. We then need to transform this into a tab-separated values (TSV) file containing passages for ColBERTv2 to index. To process all 6,945,585 articles in an efficient manner, we utilize Python's multiprocessing library and break our task into three separate processes: a dump reader process, an article reader process, and a writer process. The dump reader process, which is the main process, uses the mwparserfromhtml library to read articles from the dump file one at a time, placing them in a shared input queue. While this is happening, we spawn multiple article readers which pop articles from the queue and transform them into chunks for indexing. We extract the first paragraph from each parser using mwparserfromhtml, then prepend the article name to it to improve retrieval. If the resulting paragraph is too large for ColBERTv2 to process, we recursively break it into sentence-level chunks, making sure to prepend the title to each chunk to reduce context loss. We place all chunks in an output queue for our final process, the writer process. The writer process pops passages from the output queue, assigns them a passage id, then writes both the passage id and the passage to the TSV file. With 20 reader

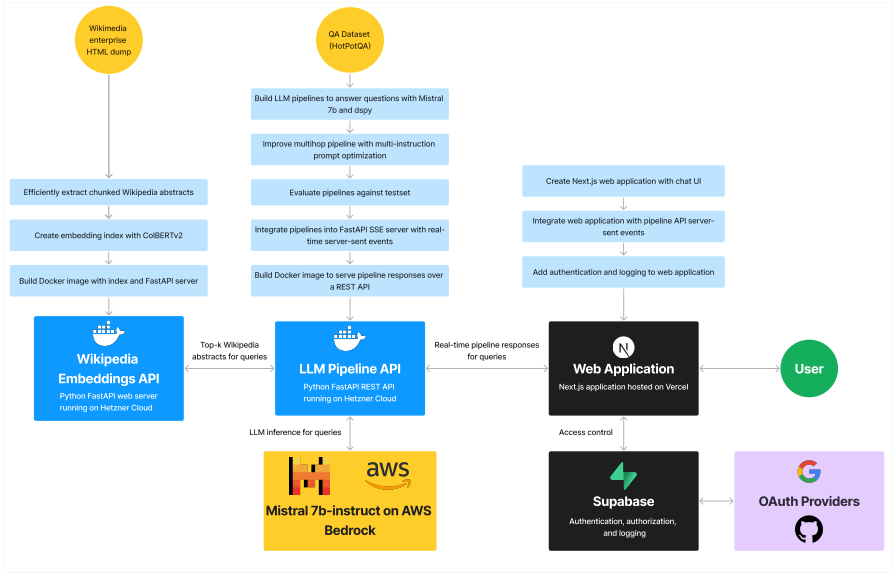

Figure 1: Architecture diagram of the system, including data sources, creation steps, and interactions.

processes, we processed the entire dump into a TSV in 3 hours and 31 minutes, or at a rate of about 548 articles/sec.

Once we have the TSV file, we can create our embedding index. We use ColBERTv2 and its Python package colbert-ai to create and store a new embedding index. The library does this by reading in the TSV file, calculating embedding centroids from a random subset of the articles, creating an embedding matrix for each article, and storing it in a file based on which centroid it is closest to.

Finally, we transform our index into a web server to serve the top-k documents relevant to a given search query. We use the basic web server provided in the colbert-ai repository and build a Docker image with it along with our index and passages. When ran, the image efficiently retrieves and serves documents from the /api/search endpoint.

4.2 LLM Pipeline Creation

We experiment with creating different LLM and RAG pipelines using the dspy library. For our retrieval model, we use the ColBERTv2 image created in the previous section. We use Mistral's 7b-instruct model as the language model for our pipelines. We chose this model since it is comparatively lightweight and cheap.

We first create pipelines which do not use any retrieval methods to establish a baseline. This includes just passing the query to the language model and using

a question-answer prompt. We also make a pipeline that uses chain-of-thought prompting, which has been shown to improve language model reasoning abilities (Wei et al., 2023).

Our next pipeline implements basic RAG to answer questions. We pass the user's query to our ColBERTv2 endpoint to get the top-10 most relevant documents. We then pass the contents of these documents directly to the chain-of-thought question-answer prompt within an added context field. We call the language model with this prompt and return back the generated answer.

We then create a multi-hop RAG pipeline based off of Khattab et al., 2021's Baleen system. We create a loop where we generate a search query given the context and the original query, retrieve three similar documents from the retrieval model, generate a summary of relevant information given the documents and the original query, and store the summary. We perform this loop twice, then pass both summaries and the original query into a chain-of-thought question-answer prompt. We pass this prompt into the language model to get our final answer. This method should allow us to answer more complex questions that require information from multiple distinct sources.

Our final pipeline brings demonstration back into the DSP framework. The dspy library provides a multi-prompt instruction optimizer (MIPRO) to improve pipeline performance. This technique uses another model to generate new prompts for each step of the pipeline. MIPRO also generates demonstrations for each step of the pipeline to use in the prompts. It then searches for the best combination of prompts and demonstrations using Bayesian analysis to optimize our metric of choosing. For this optimization, we used the HotPotQA dataset for multi-hop question answering (Yang et al., 2018). We aim to optimize average recall. We measure average recall across unigrams and longest-common-subsequences between the correct answer and the generated answer. This is inspired by the ROUGE metric from Lin, 2004 which uses subsequence precision and recall to evaluate generated summaries. We use the MIPRO optimizer along with 200 question-answer pairs and our average recall metric to create new instructions and demonstrations for the multi-hop pipeline.

4.3 LLM Inference Server

We create a web server using Python and FastAPI to serve each of these pipelines from a REST API. To improve observability over the pipelines, we want to return back information about each step of the pipeline, specifically the requests and responses of the language and retrieval models. We also want to stream this information back to the client in real time as the pipeline is running. We use the sse-starlette library to emit these events back following the server-sent events specification. However, this requires that all our values are emitted from an asynchronous generator, and dspy is an synchronous library. Therefore, we hack around the library and implement custom language and retrieval model clients that take in a reference to a synchronous thread-safe queue as a parameter. Whenever these clients receive a request, they write the request to the queue before calling the underlying model. They also write to the queue upon receiving

a response.

When a new request comes in, we create a new thread-safe queue for writing responses to. We then pass our pipeline of choice, the user query, and a reference to the queue's synchronous interface into our wrapper function and turn it into an asyncio task to run off the main thread. The wrapper function initializes the custom language and retrieval models with the output queue and then calls the pipeline with the query. Finally, it puts the final result into the output queue and also sends back a sentinel value to indicate that the pipeline is finished. Meanwhile on the main request thread, an asynchronous generator waits for new outputs to come in from the queue and emits them to the sse-starlette EventSourceResponse helper. Once the sentinel value is reached, the processes are cleaned up and the request is completed.

We deploy this server by creating a Docker image with all the required files and dependencies for running the server and the pipelines. Both the LLM inference server and the retrieval model server are running on the same machine in the cloud.

4.4 Web Application

We build a web application for users to use to interact with our LLM inference server. We use Next.js as our Javascript framework and Supabase as our database and authentication solution. The Next.js app has a home page, a login page, and a dashboard with a page for each of the LLM pipelines. The dashboard is protected from non-authenticated users. We also protect our LLM inference server behind a Next.js API endpoint that validates that the user is logged in. This endpoint also logs the request to a table in Supabase and blocks the request if the user has sent over 100 requests in the past day.

The dashboard has a sidebar with a link to pages for each pipeline. Each page has the same chat component with a text box to receive a question. Upon submission, the page calls the API and creates a new EventSource on the page. Whenever a new event is received from the API, it inserts a new chat bubble into the UI styled based on the event type. It does this until it receives a message with the property 'to: client', which indicates the the connection can be closed.

The web application is hosted on Vercel and the Supabase instance is hosted on Supabase's free cloud platform.

5 Pipeline Evaluation

We evaluate these pipelines to prove that they give better question-answering performance than the base language model, Mistral 7b-instruct. We evaluate our six pipelines, which includes the base language model, against the HotPotQA dataset for multi-hop-question answering (Yang et al., 2018). We use an evalset of 200 question-answer pairs and we repeat our evaluation three times to account for variations in the output.

Our main metric is ROUGE score as proposed in Lin, 2004. We calculate precision, accuracy, and F1 scores with ROUGE-1 (across unigrams), ROUGE-2 (across bigrams), and ROUGE-L (across the longest common subsequence). ROUGE-N scores between a candidate passage C and a reference passage R, where N is the size of the n-gram, are calculated with the following equations[1] (Lin, 2004):

Precision:

$$\text{ROUGE-N Precision} = \frac{\sum_{gram_n \in R} \text{Count}_{\text{match}}(gram_n)}{\sum_{gram_n \in C} \text{Count}_{\text{cand}}(gram_n)} \tag{1}$$

Recall:

$$\text{ROUGE-N Recall} = \frac{\sum_{gram_n \in R} \text{Count}_{\text{match}}(gram_n)}{\sum_{gram_n \in R} \text{Count}_{\text{ref}}(gram_n)} \tag{2}$$

F1-Score:

$$\text{ROUGE-N F1-Score} = \frac{2 \times (\text{ROUGE-N Precision} \times \text{ROUGE-N Recall})}{\text{ROUGE-N Precision} + \text{ROUGE-N Recall}} \tag{3}$$

where:

- $\text{Count}_{\text{match}}(w_i)$ is the number of times that n-gram $gram_n$ appears in both the candidate passage C and reference passage R.
- $\text{Count}_{\text{cand}}(w_i)$ is the number of times that n-gram $gram_n$ appears in the candidate passage C.
- $\text{Count}_{\text{ref}}(w_i)$ is the number of times that n-gram $gram_n$ appears in the reference passage R.

ROUGE-L scores are calculated with the following equations (Lin, 2004):

Precision:

$$\text{ROUGE-L Precision} = \frac{LCS(C, R)}{|C|} \tag{4}$$

Recall:

$$\text{ROUGE-L Recall} = \frac{LCS(C, R)}{|R|} \tag{5}$$

F1-Score:

$$\text{ROUGE-L F1-Score} = \frac{(1 + \beta^2) \cdot \text{ROUGE-L Precision} \cdot \text{ROUGE-L Recall}}{\beta^2 \cdot \text{ROUGE-L Precision} + \text{ROUGE-L Recall}} \tag{6}$$

where:

[1] Lin, 2004 provides a formal definition for the ROUGE-L recall metric across a single candidate passage and a set of reference passages. These equations have been derived from this definition for use with a single candidate passage and a single reference passage.

- $LCS(C, R)$ is the length of the longest common subsequence between the candidate summary C and the reference summary R.
- $|C|$ is the length of the candidate summary.
- $|R|$ is the length of the reference summary.
- β is a parameter that balances the weight between precision and recall. We use a standard value of $\beta = 1$.

6 Results

Table 1 shows our average precision, recall, and F1 scores across unigrams. All of the pipelines with retrieval (basic RAG, multihop RAG, optimized multihop RAG) outperform the base model in average precision, recall, and F1 score. The basic RAG pipeline has the highest average recall of 0.475023. While it had a slightly lower recall of 0.433750, the optimized multihop RAG pipeline has the highest average precision and F1 score of 0.366446 and 0.366634 respectively.

Table 1: HotPotQA Average ROUGE-1 Scores (200 Questions, 3 Trials)

Model	Precision	Recall	F1 Score
Optimized Multihop RAG	**0.366446**	0.433750	**0.366634**
Multihop RAG	0.295117	0.406931	0.306676
Basic RAG	0.205702	**0.475023**	0.244970
Chain of Thought QA	0.102119	0.297366	0.135585
Basic QA	0.068533	0.362222	0.099803
Unprompted mistral-7b-instruct	0.033505	0.382940	0.057758

Table 2 shows our average precision, recall, and F1 scores across bigrams. Once again, all of our pipelines with retrieval outperform the base model in average precision, recall, and F1 score. The basic RAG pipeline again has the highest average recall of 0.255855, and the optimized multihop RAG pipeline has the highest average precision and F1 score of 0.198530 and 0.204300 respectively.

Table 2: HotPotQA Average ROUGE-2 Scores (200 Questions, 3 Trials)

Model	Precision	Recall	F1 Score
Optimized Multihop RAG	**0.198530**	0.235242	**0.204300**
Multihop RAG	0.177878	0.228679	0.182876
Basic RAG	0.125862	**0.255855**	0.143879
Chain of Thought QA	0.050330	0.140313	0.063993
Basic QA	0.034693	0.183968	0.046964
Unprompted mistral-7b-instruct	0.012529	0.181804	0.021674

Table 3 shows our average precision, recall, and F1 scores across longest common subsequences. We see the same pattern: all retrieval models outperform the base model in all three metrics, the basic RAG pipeline has the highest average recall (0.474273), and the optimized multihop RAG pipeline has the highest average precision (0.365970) and F1 score (0.365968).

Table 3: HotPotQA Average ROUGE-L Scores (200 Questions, 3 Trials)

Model	Precision	Recall	F1 Score
Optimized Multihop RAG	**0.365970**	0.432639	**0.365968**
Multihop RAG	0.295047	0.406375	0.306553
Basic RAG	0.205516	**0.474273**	0.244673
Chain of Thought QA	0.101428	0.295588	0.134663
Basic QA	0.068056	0.360306	0.099051
Unprompted mistral-7b-instruct	0.033299	0.380801	0.057384

7 Discussion

The results show that the RAG pipelines which we created perform better than the underlying language model at factually answering questions. However, we find it interesting that the basic RAG model outperformed the optimized multihop RAG model in recall across all ROUGE variations. When inspecting the steps taken by each pipeline with our web application, we noticed that there were some instances in which the wrong pieces of information were being used by the optimized multihop RAG pipeline. For example, when asking it about the birthday of the 2023 Pittsburgh Steelers head coach, the pipeline fetches the correct article about the Steelers which contains their head coach. However, the pipeline also fetches an article about the head coach of the Kobe Steelers rugby team. In the summarizing step, the pipeline chooses to focus on the head coach of the Kobe Steelers, Dave Rennie, instead of the head coach of the Pittsburgh Steelers. This messes up the rest of the pipeline as it forms the next query around getting more information about the head coach of the Kobe Steelers. Once we reach the step where the final answer is generated, the model returns back "There is no answer for the question as the context does not provide information about Dave Rennie coaching the Pittsburgh Steelers." As we add more steps to the pipeline, the likelihood that the underlying language model fails to perform the intended task in at least one step increases. Thus, we attribute this difference in recall to the higher chance of failure due to the increase in steps. However, across all ROUGE variants, the increase of average precision and F1 score from using optimized multihop RAG is greater than the decrease of precision when compared to basic RAG. For this reason, we consider the optimized multihop RAG to be the higher-quality model.

We are also pleased to see that MIPRO optimization improved all statistics across all ROUGE variants. While we were trying to optimize for average recall,

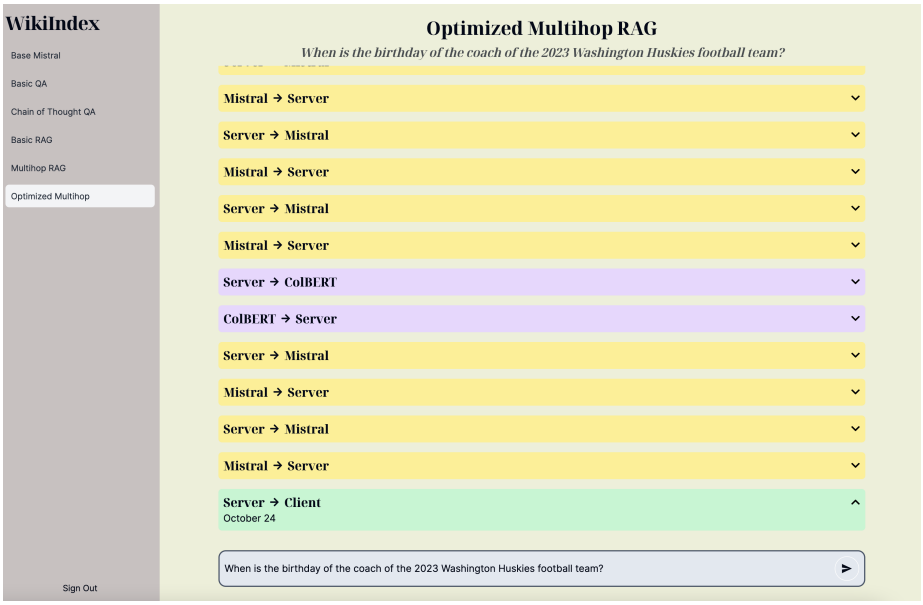

Figure 2: Screenshot of the web application returning a response for the query "When is the birthday of the coach of the 2023 Washington Huskies football team?"

we also increased average precision and F1 score across all ROUGE variants by a significant amount. We attribute this to the optimized prompts, along with the generated demonstrations for each step.

Finally, we fully deployed the web application and demoed it during the Cal Poly College of Engineering Senior Project Expo. The web application is fully functional and provides insight into how the pipelines operate under the hood. We also found it useful as a visual tool to understand when and why RAG pipelines fail to correctly answer questions. We received positive feedback about how the steps are streamed back along with the performance of the optimized multi-hop RAG pipeline. The application is currently deployed at https://wikiindex.vercel.app[2].

8 Future Work

One of the areas we want to look into is expanding the amount of information we use in our retrieval model. Currently, we only use the first paragraph of each Wikipedia article. The first paragraph contains important information, but it may not contain all the information needed to answer specific queries. We experimented with using the full text of each article but it resulted in an

[2]We will probably be taking it down in around a month to save on cloud costs.

index that required over 20 gigabytes of memory to load for searching. We want to look into methods of distributing these indexes across multiple servers so we can search across a larger amount of data without also needing to scale compute on a single machine. This would also open us up to including other sources of data, such as tabular data.

We also want to look into how using a different language model changes performance. We chose Mistral 7b-instruct due to its cost-effectiveness and its ability to follow directions. We also wanted to use it to show that we could improve its base performance. As language models have improved over time, we wonder if this same approach would also show performance increases in a newer model.

Finally, we want to see how these pipelines perform using out-of-domain data. While we don't know the exact dataset that was used to train Mistral 7b-instruct, we assume that it contained some of the same facts that were present in our Wikipedia abstracts. If we use a specific out-of-domain dataset, such as technical documentation for a product, we could see even larger performance gains as the base model would not be able to accurately answer questions about information that it was not trained on.

References

Bhattacharyya, M., Miller, V. M., Bhattacharyya, D., & Miller, L. E. (2023). High rates of fabricated and inaccurate references in chatgpt-generated medical content. *Cureus*, *15*(5).

Hu, K. (2023). Chatgpt sets record for fastest-growing user base - analyst note. *Reuters*. https://www.reuters.com/technology/chatgpt-sets-record-fastest-growing-user-base-analyst-note-2023-02-01/

Khattab, O., Potts, C., & Zaharia, M. (2021). Baleen: Robust multi-hop reasoning at scale via condensed retrieval. *CoRR*, *abs/2101.00436*. https://arxiv.org/abs/2101.00436

Khattab, O., Santhanam, K., Li, X. L., Hall, D., Liang, P., Potts, C., & Zaharia, M. (2023). Demonstrate-search-predict: Composing retrieval and language models for knowledge-intensive nlp.

Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W.-t., Rocktäschel, T., Riedel, S., & Kiela, D. (2020). Retrieval-augmented generation for knowledge-intensive nlp tasks. In H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, & H. Lin (Eds.), *Advances in neural information processing systems* (pp. 9459–9474, Vol. 33). Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2020/file/6b493230205f780e1bc26945df7481e5-Paper.pdf

Lin, C.-Y. (2004). ROUGE: A package for automatic evaluation of summaries. *Text Summarization Branches Out*, 74–81. https://aclanthology.org/W04-1013

Ovadia, O., Brief, M., Mishaeli, M., & Elisha, O. (2024). Fine-tuning or retrieval? comparing knowledge injection in llms.

Santhanam, K., Khattab, O., Saad-Falcon, J., Potts, C., & Zaharia, M. (2022). Colbertv2: Effective and efficient retrieval via lightweight late interaction.

Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., Chi, E., Le, Q., & Zhou, D. (2023). Chain-of-thought prompting elicits reasoning in large language models.

Yang, Z., Qi, P., Zhang, S., Bengio, Y., Cohen, W. W., Salakhutdinov, R., & Manning, C. D. (2018). HotpotQA: A dataset for diverse, explainable multi-hop question answering. *Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

Zhao, W. X., Zhou, K., Li, J., Tang, T., Wang, X., Hou, Y., Min, Y., Zhang, B., Zhang, J., Dong, Z., Du, Y., Yang, C., Chen, Y., Chen, Z., Jiang, J., Ren, R., Li, Y., Tang, X., Liu, Z., ... Wen, J.-R. (2023). A survey of large language models.