Nicholas Lombardo (nrl44)

Introduction

In the previous parts of this project, we designed a retrieval-augmented generation (RAG) pipeline to enhance a large language model's responses using information stored in a document database. A JSON collection of documents was processed, with each document embedded as a 768-dimensional vector of floating-point values. At query time, user input is similarly embedded into a 768-dimensional vector, and the system retrieves the three nearest neighbor vectors to identify the most relevant documents. These retrieved documents are then supplied as context to the language model for response generation. In this report, we analyze the system's performance, identify key bottlenecks, and explore optimization techniques to improve the efficiency and effectiveness of the RAG pipeline.

Current Design:

In the unmodified system design, the LLM generation step accounted for by far the largest portion of the total runtime, followed by the query encoding step. This behavior aligns with expectations, as even though the TinyLlama model is small by LLM standards, it still contains on the order of one billion parameters and is executed locally, which inherently incurs significant computational cost. Additionally, the model must be run repeatedly to generate each new token, and each response consists of several hundred generated tokens, further increasing latency. The next most time-consuming component was the query encoding step. Although the encoder model is much smaller than the LLM, it is still substantially larger than the remaining components of the pipeline, making it the second largest contributor to overall runtime.

	Prompt 1	Prompt 2	Prompt 3	Prompt 4	Prompt 5	Average (ms)	Standard Dev (ms)
Query Encoding	846.02	65.4	81.35	90.88	83.53	233.436	342.5709517
Vector Search	19.14	1.1	1.07	1.11	1.05	4.694	8.075594715
Prompt Augmentation	0.02	0.01	0.01	0.01	0.01	0.012	0.0044721359
LLM Generation	11870.46	11623.01	10408.03	10057	2073.11	9206.322	4061.664708

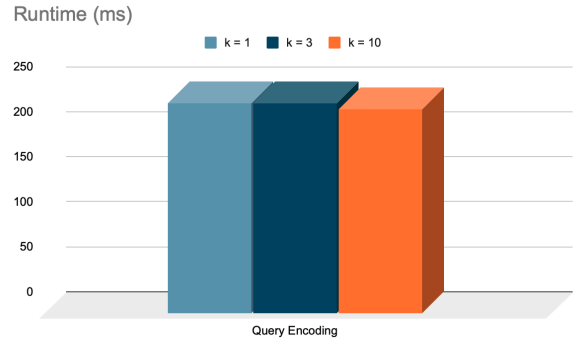
This aggregated data ran on five different prompts reveals that the LLM Generation (97.5%) step is the bottleneck of our program, followed by the Query Encoding (2.5%), Vector Search (0.05%), and finally the Prompt Augmentation step (0.0001%).

kNN Optimization

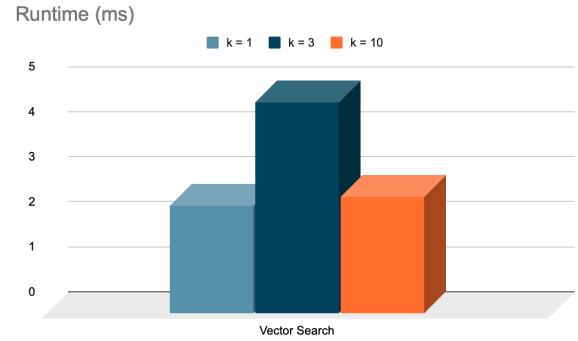
To explore the impact of using different numbers of k-nearest neighbors in our system, we evaluate two alternative configurations: k=1and k=10, and compare them against the control

k=3. For each configuration, we analyze both the system's runtime and the performance of the LLM to understand how varying k affects efficiency and response quality.

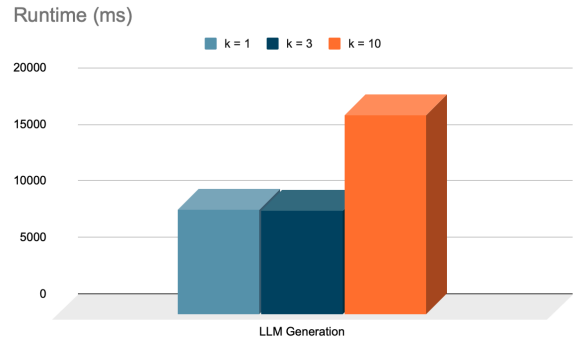
Query Embedding



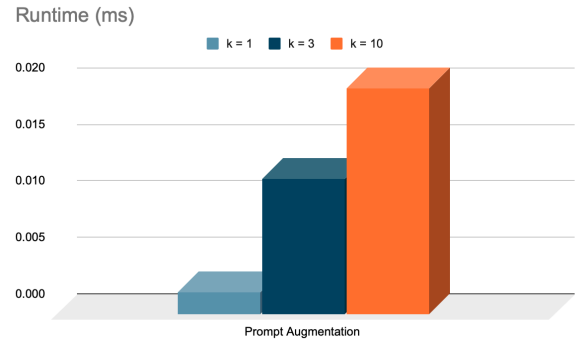
Vector Search



LLM Generation



Prompt Augmentation



K=1	Prompt 1	Prompt 2	Prompt 3	Prompt 4	Prompt 5	Average (ms)	Standard Dev (ms)
Query Encoding	851.96	78	82.64	76.92	78.04	233.512	345.72997
Vector Search	7.84	1.04	1.03	1.03	1.04	2.396	3.0432926
Prompt Augmentation	0.01	0	0	0	0	0.002	0.0044721
LLM Generation	9273.48	9054.75	9495.77	9232.86	9202.79	9251.93	159.35864

	Prompt 1	Prompt 2	Prompt 3	Prompt 4	Prompt 5	Average (ms)	Standard Dev (ms)
Query Encoding	846.02	65.4	81.35	90.88	83.53	233.436	342.570951
Vector Search	19.14	1.1	1.07	1.11	1.05	4.694	8.07559471
Prompt Augmentation	0.02	0.01	0.01	0.01	0.01	0.012	0.00447213
LLM Generation	11870.46	11623.01	10408.03	10057	2073.11	9206.322	4061.66470

K= 10	Prompt 1	Prompt 2	Prompt 3	Prompt 4	Prompt 5	Average (ms)	Standard Dev (ms)
Query Encoding	816.51	80.11	81.46	78.92	78.08	227.016	329.53962
Vector Search	8.46	1.1	1.12	1.14	1.11	2.586	3.2836991
Prompt Augmentation	0.02	0.02	0.02	0.02	0.02	0.02	0
LLM Generation	15008.52	21289.4	21832.1	17204.07	13121.65	17691.148	3821.2874

As expected, the k = 1 configuration exhibited the shortest total runtime, while k = 10 resulted in the longest. The most pronounced difference occurred in the LLM generation step, which experienced a substantial speedup as k decreased. This is likely because the prompt passed to the LLM grows significantly with larger k, containing ten sentences of retrieved context instead of three or one, which increases both input processing and token generation time. As a result, the majority of the runtime improvement can be attributed to this stage. The prompt augmentation runtime likewise also increased as k increased as the prompt concatenation step in theory grows linearly with k as more strings are added. In practice, the effect of this step on the total runtime of the LLM is negligible.

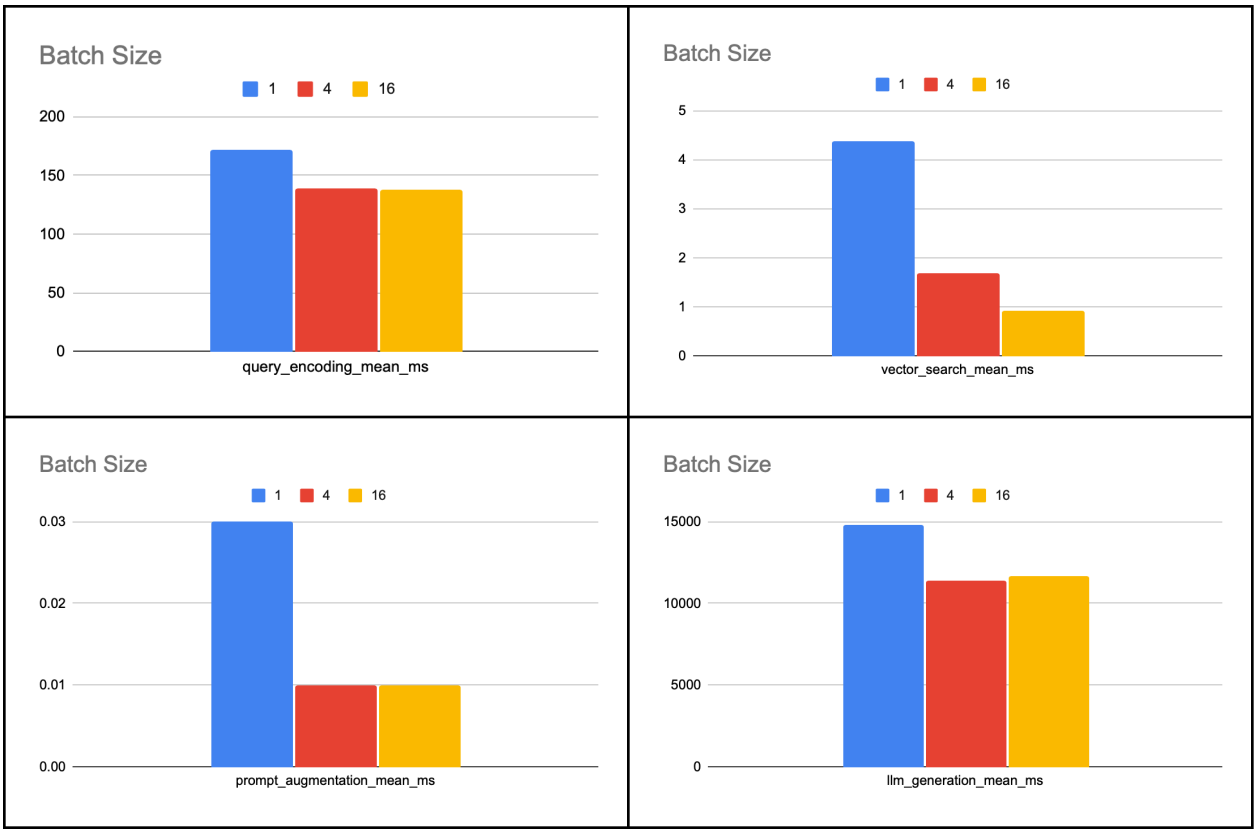
The remaining pipeline components showed no statistically significant change across the three configurations. This was somewhat surprising, particularly for the vector search step, as we initially expected a noticeable speedup when retrieving a single neighbor compared to ten; however, in practice, its runtime remained effectively constant.

Two things stand out as interesting from this data, the first is the query embedding time for the first prompt is always significantly longer than the others. I believe this is caused from the embedding model booting up the first time it is used, and then gets cached which causes the subsequent uses to be much quicker. The other thing that stands out to me is the outlier that occurred in k = 3 for the prompt: who is the president? The answer to this question from the LLM was incredibly brief and said the answer was Joe Biden (as the RAG database is likely not updated). Since the response was so quick the LLM generation step was incredibly brief which skewed the data.

Lastly, it is clear from the LLM responses that k = 10 is the weakest configuration. In addition to having the longest runtime, it consistently produced the most incoherent and nonsensical outputs. For example, when asked about renewable energy, the model responded with a meaningless repetition such as “The potential energy in the sun is 1000x1000x1000x1000x1000x1000…”. This behavior likely occurred because many of the retrieved sentences were not directly relevant to the query, causing the model to become confused by excessive and noisy context. In contrast, the k = 1 and k = 3 configurations produced responses of comparable quality, but k = 3 was the only setting that consistently returned a correct answer to the question “who is the president?” by identifying Joe Biden.

Batch Size Optimization

Going forward, instead of using five manual queries, I will use the queries.json file and measure the time each configuration takes to process all 100 queries. To test the optimal batch size, I will evaluate batch_size = 1, 4, and 16.



The results show that the largest and most dependable improvement from batching occurs in the vector search step. Increasing the batch size from 1 to 4 and then to 16 reduces the average vector search time from about 4.38 ms per query to 1.68 ms and then to under 1 ms, which aligns with how FAISS benefits from processing multiple queries together.

The other components also show decreases in average time, including both query encoding and LLM generation. However, these improvements are harder to interpret. The variance in these steps is high, especially for LLM generation, so it is difficult to tell whether the reductions are caused by batching or simply by statistical noise in the measurements. In other words, the numbers look better, but the underlying data does not allow a confident conclusion.

Overall, batching most clearly helps vector search, while the apparent gains in the other components cannot be reliably attributed to batching based on this dataset.

The raw data (includes variances):

batch_size	total_queries	query_encoding_mean_ms	query_encoding_std_ms	vector_search_mean_ms
1	100	171.45	170.01	4.38
4	100	139.51	177.48	1.68

16	100	137.42	170.35	0.92

vector_search_std_ms	prompt_augmentation_mean_ms	prompt_augmentation_std_ms	llm_generation_mean_ms	llm_generation_std_ms
5.36	0.03	0.04	14805.94	8055.14
0.8	0.01	0	11400.51	4246.96
0.52	0.01	0	11680.45	4431.23

Different Size Encoder Models

For this test we experimented with three encoder models: BAAI/bge-base-en-v1.5 with 768 dimensions, sentence-transformers/all-MiniLM-L6-v2 with 384 dimensions, and BAAI/bge-large-en-v1.5 with 1024 dimensions.

model_name	embedding_dim	avg_query_encoding_ms	avg_vector_search_ms	avg_prompt_augmentation_ms	avg_llm_generation_ms	avg_total_pipeline_ms
BAAI/bge-base-en-v1.5	768	181.26	2.86	0.02	9536.44	9720.57
sentence-transformers/all-MiniLM-L6-v2	384	35.15	2.34	0.02	12997.81	13035.31
BAAI/bge-large-en-v1.5	1024	480.54	4.1	0.02	11643.12	12127.78

As expected, the larger embedding models produced longer query encoding times and slightly slower vector search times, since both operations scale with embedding dimensions. What stood out, however, was that this relationship was not linear. The 384-dimensional model had less than half the latency of the 768-dimensional model despite being only half the size. The smallest model also ended up with the longest LLM generation time, which resulted in the highest total pipeline latency. This may be due to the model retrieving less useful or noisier context at k = 3, causing the LLM to spend more time processing the augmented prompt. The prompt augmentation step was effectively the same for all three models.

When regenerating the embeddings for preprocessed_documents.json, the largest model took significantly longer than the others. This reinforces that model size impacts not only online query latency but also the offline preprocessing time required to encode the full document set.

Also notable is that the mini model performed worse at retrieving relevant sentences, especially for higher values of k, which led to more irrelevant context being passed to the LLM. This may explain why its LLM generation time was the longest, although further analysis would be needed to confirm this.

In general the medium and large encoding models had similar LLM output with the smaller model having a slightly less accurate response of what was in the RAG database.

Conclusion

The evaluation shows that LLM generation overwhelmingly dominates the pipeline's runtime, with query encoding as the only other substantial contributor. Adjusting k and batching offered the most noticeable improvements, with batching reliably speeding up vector search and smaller k values reducing LLM workload. Experiments with different encoder sizes highlighted a clear tradeoff: smaller models encode faster but can retrieve weaker context, while larger models increase preprocessing time without guaranteeing better downstream performance. Overall, effective optimization in a RAG system depends on balancing retrieval quality with computational cost rather than simply scaling components up or down.