Executive Summary

This project developed and evaluated a complete Retrieval-Augmented Generation system for question-answering using the RAG Mini Wikipedia dataset. The system was implemented locally using open-source models: all-mpnet-base-v2 for embeddings (768 dimensions), FAISS for vector storage, and Flan-T5-base for generation. Systematic experimentation across prompting strategies revealed basic prompting outperformed complex approaches (F1=0.444 vs CoT's 0.064). Parameter optimization testing 6 configurations identified all-mpnet-base-v2 with top_k=1 as optimal (F1=0.632, EM=56%). Two enhancements were implemented: cross-encoder re-ranking and query rewriting. Re-ranking alone improved F1

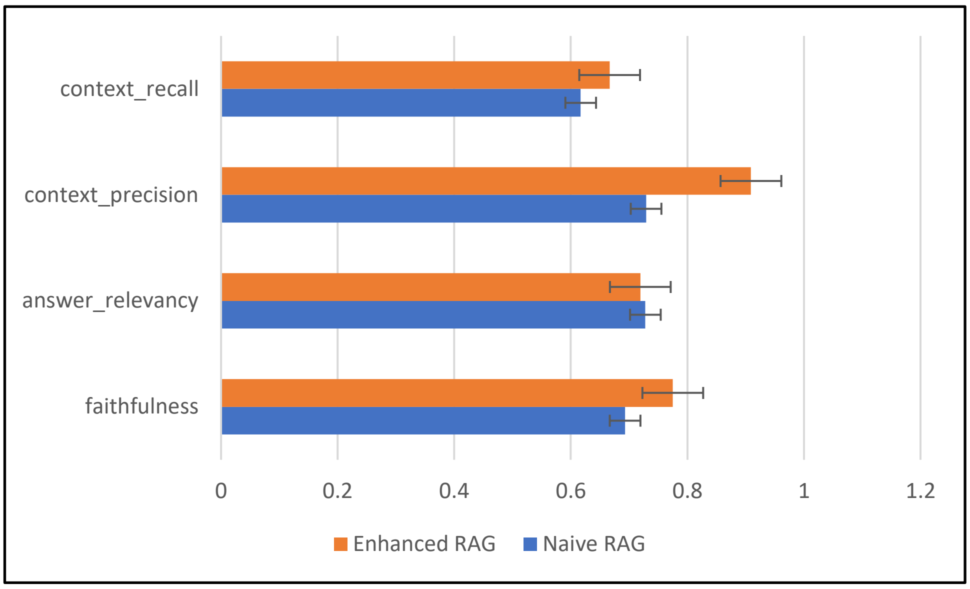


Figure 1: Comparative performance of NaiveRAG and EnhancedRAG implementation on RAGAs based evaluations

from 0.556 to 0.615 (+10.5%). RAGAs evaluation on partial data (40-44 questions per system) showed enhanced RAG improved all metrics except answer relevance. The implementation demonstrates that re-ranking provides measurable improvements in retrieval precision, though evaluation was limited by API constraints and strict word-overlap metrics that penalized verbose but accurate responses.

System Architecture: How was this RAG system implemented?

This exercise has followed the structure shown below. The development of the pipeline, as well as notebooks – followed by the evaluations were all done locally. New environments in python were created for the assignment itself as well as the RAGAs evaluations. This was done to ensure quick and clean runs, and while setup had hiccups – it wasn't messy. The decision to do this locally as well as model choices, evaluation tests was largely based in cpu capacity, and stay indifferent to session timeouts and cloud limits – considering this process was iterative. Except disappointing RAGAs evaluation runs, everything was smooth and worth the trade-off.

Configurations and requirements were laid out separately as is noted below – using these, pipelines were created. Using notebooks, pipelines were used and improved. The results were generated and saved separately using notebooks. Considering the production-like design expected, efforts were made to ensure the reproducibility of this exercise. A git workflow was initiated and adhered to.

For naïve RAG, sentence-transformers like all-MiniLM-L6-v2 and all-mpnet-base-v2 were used. The baseline is measured using all-mpnet-base-v2, and this was due to it's better performance upon initial exploration as has been documented in the evaluations. Later implementations of RAG (naïve and enhanced) used just the better all-mpnet-base-v2 model. While this wasn't efficient considering local deployment, it was effective. FAISS was the vector DB used throughout the exercise. The inner product for cosine similarity with normalization was used for indexing in the Vector DB via FAISS. Retrieval was executed by semantic search with varying top_k iterations. The generation of answers post retrieval was done using Google's Flan T5-base model. All these models were retrieved from HuggingFace. The model choice was made due to it's manageable size with local inference and zero API costs. Experimentation with embedding dimensions = 384, 768 and top_k retrieval were conducted as has been documented in the evaluations. The data this pipeline has been tested over is the RAG-MiniWikipedia dataset available on HuggingFace.

The exercise's architecture looks like this:

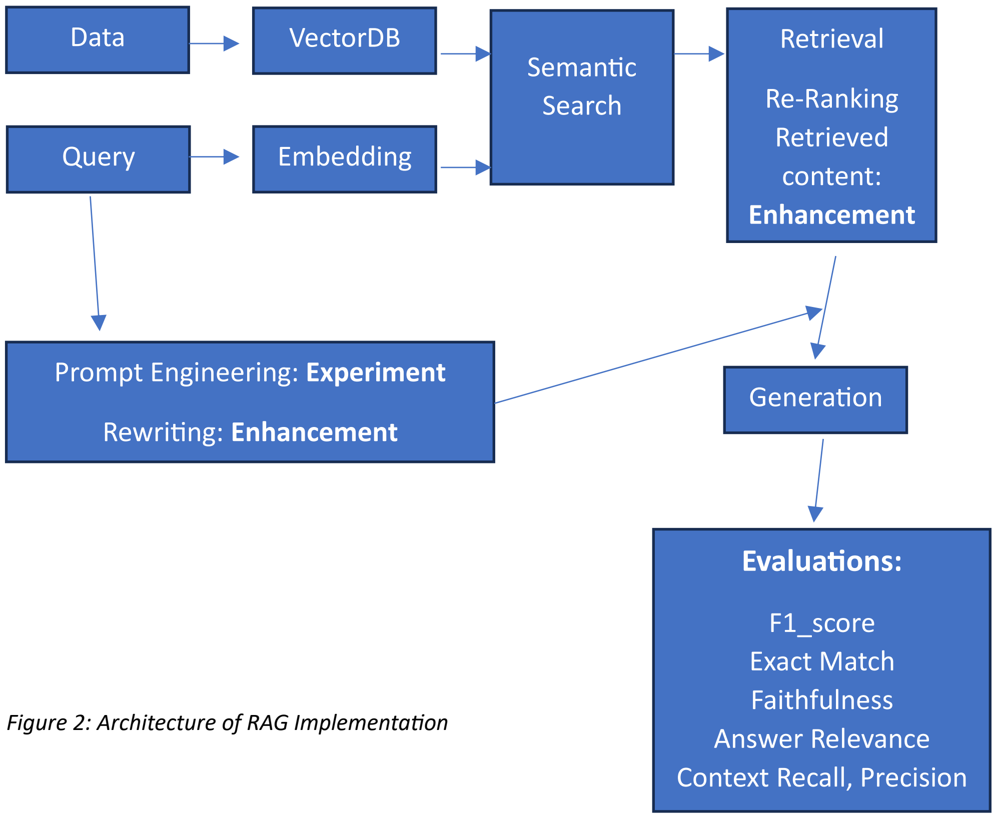


Figure 2: Architecture of RAG Implementation

Experimenting with NaïveRAG: Deciding what works

Once NaïveRAG was implemented, a test set of the first 10 q/a pairs was worked with to identify capabilities – how well it performed, and how long it took, and if there were any issues with the implementation. The implementation was then experimented with. Using the same pipeline and capabilities, performance was tested for top 100 questions using 4 prompting strategies. These experiments yielded, in full irony, 'basic' prompting as the best strategy. Basic prompting was simply the question as is. Basic prompting yielded an average 0.44 as the F1 score. 39%, or 39 questions were answered *exactly* as the ground truth. Chain of thought prompting was the question, concatenated after "Let's think through to the answer in a step by step manner.". This strategy yielded 0.064 as the average F1 score. Due to the F1 calculations as the intersection of words in the predicted answer versus ground truth, chain of thought lost out on precision. Chain of thought prompting usually gives back verbose answers, which are comparatively deeper in knowledge – but this doesn't help in our case, given our F1 score calculations. Similarly, exact matches were 0. The string, "You are an expert researcher with great depth in your knowledge about this topic." was concatenated to the query in the 'persona' prompting strategy. This confirming persona assignment yielded an average 0.40 F1 score, and 35 out of 100 exact matches. In the fourth experiment, the LLM was explicitly instructed to be truthful, precise and deliberate in it's answers with the string, "Instructions: Based on the provided context, answer the question accurately and concisely. If the answer is not in the context, say so. Do not answer for the sake of answering, say no if the information is unavailable." concatenated pre-question. This yielded 0.26 as the average F1 score and 20 exact matches across 100 questions.

Strategy	**Avg F1 Score**	**Exact Match %**	**Time (min)**
basic	0.444	39.00%	1.8
cot	0.064	0.00%	8.3
persona	0.403	35.00%	1.5
instruction	0.267	20.00%	1.3

Table 1: Comparative performance of prompting strategies over NaïveRAG

Further experimentation was done over embedding dimension options, and with different retrieval rules. All-MiniLM-L6-v2 is a 384 dimension embedding model. Using sentence-transformer all-mpnet-base-v2, a 768 dimension iteration was also completed. For each iteration, three different retrieval rules were implemented – top_k = 1, 3, 5. These configurations were tested with a dataset of 50 questions. Average F1 scores and exact matches were used to compare these configurations. The experimentation yielded an interesting insight. For the smaller dimension iteration, top_1 yielded the best performance at average F1 score = 0.62. It was a 768 dimension iteration and top_k = 1 combination that yielded the best F1 score on average, at 0.63. The next best iteration for 768 dimensions was top_k = 3, which yielded an average F1 score of 0.55. This combination was finalized as the baseline – to compare enhancements to. This decision was made to allow enhancements like re-ranking and query re-writing on a comparative base.

Embedding Model	**Top-K**	**Avg F1**	**Exact Match %**
all-MiniLM-L6-v2	1	0.516	46.00%
all-MiniLM-L6-v2	3	0.62	58.00%

all-MiniLM-L6-v2	5	0.423	40.00%
all-mpnet-base-v2	1	0.632	56.00%
all-mpnet-base-v2	3	0.556	50.00%
all-mpnet-base-v2	5	0.343	32.00%

Table 2: Comparative performance of embedding dimension-top_k combinations over NaïveRAG

Enhancing RAG

Thereafter, two specific enhancements in capabilities were made. Query re-writing and re-ranking of retrieved contexts was integrated in the NaïveRAG base. Confidence scoring and citation grounding are not performative technical improvements, but help confirm answers, explain predictions and increase interpretability for a human using the RAG system. While they are important metrics, the intention was to enhance the system technically. Due to the local application of this exercise, downloading and using a second embedding model for multi-vector retrieval was not as feasible and pragmatic as query rewriting and reranking was. Query rewriting was implemented using the generation model already loaded as part of the pipeline. Here, the prompt was refined by the generator to be more deliberate and nuanced in nature. This refined prompt and the original question became the query which was combined with the relevant retrieval as an input for final answer generation. Query rewriting implementation was fairly straightforward, with no particular challenge faced. This enhancement yielded a decent betterment in system capabilities. The average F1 score across 50 test questions (for quicker evaluation) was improved to 0.576 from 0.556, and one additional question's answer was predicted exactly like the ground truth.

Re-ranking was done using cross-encoders. Top_10 relevant contexts were retrieved and then re-ranked by the cross-encoder. The top_3 from these 10 were fed to the generation model alongside the query as context for answer prediction. This was not a tricky enhancement either, with a straightforward implementation – deriving from the pre-built pipeline and integrating the output of reranking as context just as the Naïve pipeline passed on the top 1/3/5. The additional shuffle and filter step yielded an enhanced average F1 score of 0.615 and 3 additional matched to ground truths. This was particularly a surprise, considering the previously modest improvements when top_3/ 5 retrieval was experimented with. Understandably, cross-encoded re-ranking allows a slightly better degree of semantic inference by the generation model. This means that often, there are better (objectively) answers behind the ones an LLM outputs, that simply lose out on objective metrics and don't reach us. Food for thought about the increasing standardization and the resultant sliding of the 'nuance' and 'subjectivity' of this world.[1]

Implementing both together didn't particularly increase the objective performance of the system, and therefore we could generalize that simple re-ranking has greater effect on the

[1] Justin. (2025, August 19). Is the Mid-Range Dead? Algorithmicathlete.com; The Algorithmic Athlete. https://algorithmicathlete.com/blog/is-mid-range-dead

RAG implementation's objective performance. This version was then taken as the EnhancedRAG implementation stemming from the NaïveRAG exercise. These evaluations are stored in the results folder.

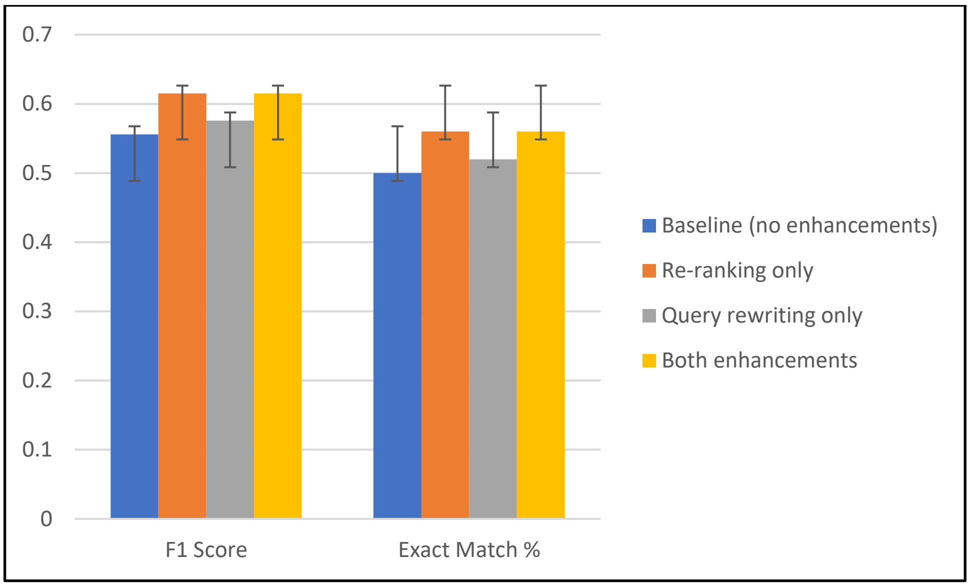


Figure 3: Comparative performance improvement of 2 enhancements at once, and together

Evaluating enhancements v. NaïveRAG implementations – using RAGAs

Finally, holistic evaluations were implemented for the NaïveRAG system, as well as the enhanced system (with re-ranking). RAGAs, an evaluation framework for RAG systems was used. OpenAIs gpt-4o-mini was used to conduct these evaluations. These evaluations were conducted over 50 test questions from the RAG-mini-Wikipedia dataset. These 50 datapoints were tested for a) faithfulness, or the grounded-ness and explicit presence of predicted answers in the data, b) answer relevance, or the relevance of the answer to the original query, c) context precision, or the ranking of relevant contexts higher than less relevant contexts by the retriever and d) context recall, or the portion of true relevant retrievals that are actually retrieved. While faithfulness and answer relevance are metrics that evaluate the generator component of the pipeline, context recall and context precision are evaluators of the retrieval bit of the RAG pipeline. The evaluation is automated over the dataset – and our evaluation ran across 44 of 50 datapoints in the NaïveRAG dataset, and 40 of the 50 from the EnhancedRAG dataset. These datasets were generated by local implementation of the pipeline. The dataset has the generated answer, query, context(relevant retrieval), ground truth, among other details.

It is notable that by enhancing the RAG pipeline, performance was lost on the generator component of the RAG pipeline, with answer relevance worsening, and faithfulness increasing by 8 basis points. It can be interpreted that answer relevance was impacted due to the addition of re-ranking and the resulting noise that ambiguously close relevant retrievals might present to the generator as input – with little to no discrimination. The faithfulness did increase with enhancements – and this can be attributed to the re-ranking resulting in very particular contexts being inputted with the query to the generator – thereby increasing it's ability to accurately predict a grounded answer.

The enhancement was on the retrieval side, and to see improvements in capabilities suggests the success of these enhancement mechanisms. While context recall increases only 4 basis points, context precision – or the ranking of highly relevant contexts higher than less relevant contexts – increased by a substantial 17 basis points. This is natural, in the fact that these contexts are already outputted by the retriever after re-ranking them for relevance, after sampling the top_10. This evaluation provides credence to the enhancement itself, but it remains to be seen (with greater depth in evaluation) if the enhancement helps the pipeline in full as such.

Considerations and Limitations of the RAG implementations

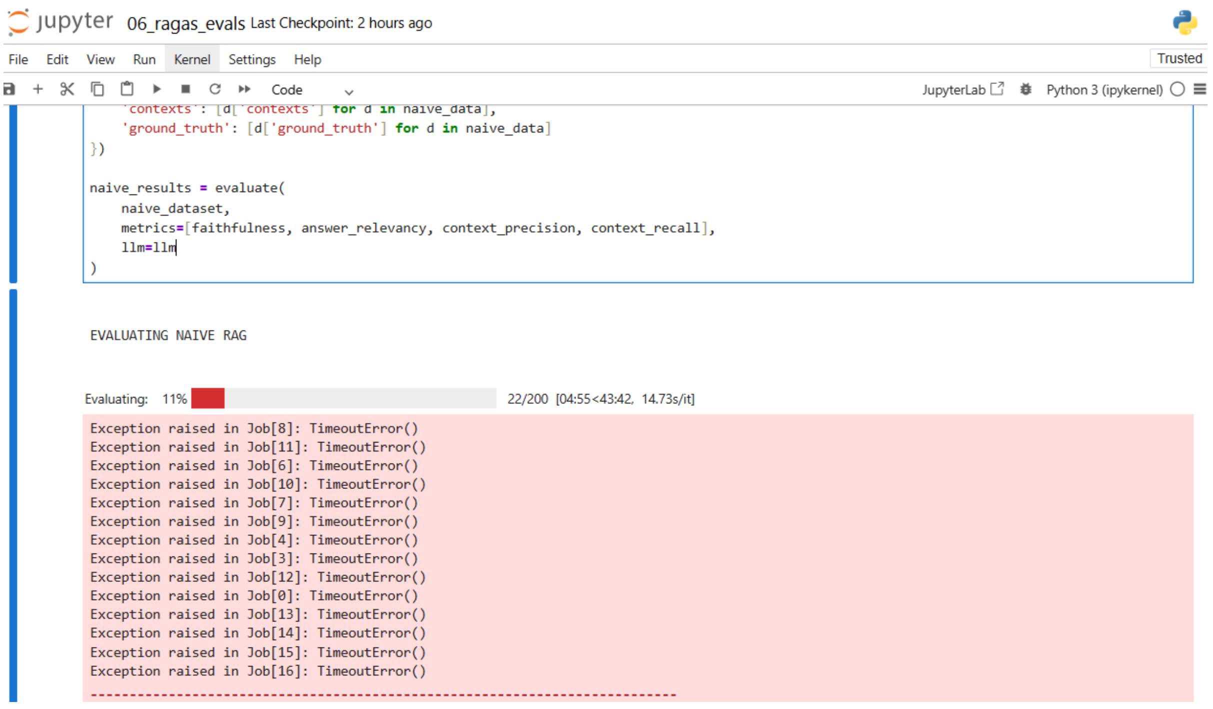
This implementation is built with a pre-planned framework for future scaling. Each functionality of the pipeline is separately defined, and tested. The implementation was phased, allowing testing and evaluating with different capabilities in the pipeline. The reproducibility of this exercise lies in the configs.py file. Baring changes to that, no particular code changes would be required to redo an iteration of this exercise – by way of the delinked application of the different functionalities. While this wasn't the case during development, and glitches like interdependent function calls were present, an effort has been made to a) remove them and decouple the 'how' form the 'what', b) introducing logging – to help identify blockers and c) descriptive documentation.

Some considerations to keep in mind for deployment are about storage and local environments. During the exercise, storage was a blocker during initial generator loading. Anaconda packages, and the embedding models already take up about half of the memory, and additionally having to load a generator model may prove to be difficult locally. Do consider cloud-based development, and therein allow advanced models to be used. RAGAs evaluations are run better at the beginning of day – but this could be a function of the API capacity constraint. Limitations lie in the evaluations of this exercise's varying experiments and enhancements. The experiments are evaluated objectively – on the predicted word's presence in ground truth and the predicted word's exact matching to the ground truth. This results in 'Yes' and 'yes' being categorized as different answers under exact match metrics. The dataset also averages only ~62 word answers, so any verbosity on part of the generator could objectively classify the generation as bad. This prevents the productive testing of prompting strategies like chain of thought.

Another limitation lies in the assessment of the enhancements. These were assessed using RAGAs as a RAG evaluation framework. This results in comparable instances – but these cannot be generalized. API limitations during RAGAs evaluation allowed 44 questions out of 50 from the Naïve data set to be evaluated, leaving out random questions and random respective metrics. As a result, these evaluations are particularly robust. They are also averages taken only from evaluated values – which might imply easier questions being overrepresented, as longer q/ a pairs may result in greater probability of TimeOutErrors due to evaluation requiring a larger number of API calls. Likewise, the Enhanced RAG infrastructure was evaluated over 40 of 50 test datapoints. Overall, while this implementation shows enhancements have worked, rigorous testing across this dataset, and using larger datasets, with more robust evaluations can help determine the real improvement over a naïve iteration.

Appendix:

RAGAs initial evaluation timeouts


jupyter 06_ragas_evals Last Checkpoint: 2 hours ago
File Edit View Run Kernel Settings Help
Trusted
Code
JupyterLab
Python 3 (ipykernel)

```
    'contexts': [d['contexts'] for d in naive_data],
    'ground_truth': [d['ground_truth'] for d in naive_data]
})

naive_results = evaluate(
    naive_dataset,
    metrics=[faithfulness, answer_relevancy, context_precision, context_recall],
    llm=llm
)
```

```
EVALUATING NAIVE RAG

Evaluating:  11%  22/200 [04:55<43:42, 14.73s/it]
Exception raised in Job[8]: TimeoutError()
Exception raised in Job[11]: TimeoutError()
Exception raised in Job[6]: TimeoutError()
Exception raised in Job[10]: TimeoutError()
Exception raised in Job[7]: TimeoutError()
Exception raised in Job[9]: TimeoutError()
Exception raised in Job[4]: TimeoutError()
Exception raised in Job[3]: TimeoutError()
Exception raised in Job[12]: TimeoutError()
Exception raised in Job[0]: TimeoutError()
Exception raised in Job[13]: TimeoutError()
Exception raised in Job[14]: TimeoutError()
Exception raised in Job[15]: TimeoutError()
Exception raised in Job[16]: TimeoutError()
---------------------------------------------------------------------------
```

AI Usage Logs - Link