Design Analysis and Algorithm (DAA) Support RAG-based Question Answering System

Course: Introduction to Text Analytics– Spring 2025

Submitted By:

- Hamna Inam Abro -27113,
- Zuha Aqib- 26106,
- Zara Masood- 26928

Submission Date: 20th April 2025

Table of Contents

1. Platform Details

The development and experimentation of this Retrieval-Augmented Generation (RAG) system was carried out using multiple platforms:

Task	Platform
Data scraping and preprocessing	Local machine (Windows), GitHub, Kaggle
Model experimentation and tuning	Kaggle
Evaluation and final reporting	Markdowns in ipynb notebook & Google Docs

Table 1-Platform Details

2. Data Details

Source:

Lecture PDFs hosted at CMU 15-451: Design and Analysis of Algorithms – Fall 2011

https://www.cs.cmu.edu/~15451-f15/lectures/ (lectures 1, 2, 3, 4, 5, 8, 9 ,11)

Corpus Size: 8 lecture PDFs

Total Pages: ~101 pages combined

Average Document Size: ~500 KB

3. Algorithms, Models, and Retrieval Methods

Experimental Setup

Retrieval Methods Employed

We implemented and tested our DAA RAG system with three retrieval strategies:

- **Semantic Search**: Leverages sentence embeddings to retrieve contextually similar chunks.
- **Keyword Search**: Uses keyword matching for exact or partial term overlaps.
- **Hybrid Search**: Combines semantic and keyword-based methods, using equal weights *(semantic_weight=0.5, keyword_weight=0.5)*.

This was to leverage the advantages of both the retrieval techniques, as semantic search offers improved understanding of contextual queries, while keyword search ensures precision. Hybrid search balances both, aiming to optimize both relevance and faithfulness

LLMs and Embedding Models

Model	Purpose	Justification
sentence-transformers/all-MiniLM-L6-v2	Embeddings	Efficient and fast for computing dense vectors for retrieval, suitable for low-latency use cases
meta-llama/Llama-3.2-1B	QA Generation	Chosen for its balance of generation quality and speed in constrained environments.
TinyLlama/TinyLlama-1.1B-Chat-v1.0	QA Generation	Lightweight model optimized for chat-style tasks; ideal for edge deployment and low-resource settings
microsoft/phi-2	QA Generation	Known for its strong reasoning capabilities in compact size; useful for accurate yet efficient answer generation

Table 2- LLMs and Embedding Models

Chunking Strategy Analysis

Chunking plays a pivotal role in RAG systems as it determines how textual context is divided before being passed into the retriever and subsequently the LLM. The strategies tested here vary across:

- **Chunk Sizes**: 250, 500, 800, 1000 tokens
- **Chunk Overlaps**: 100, 200 tokens
- **Top-k Values**: 3, 4, 5

Impact of Chunking Strategy on Relevancy Score

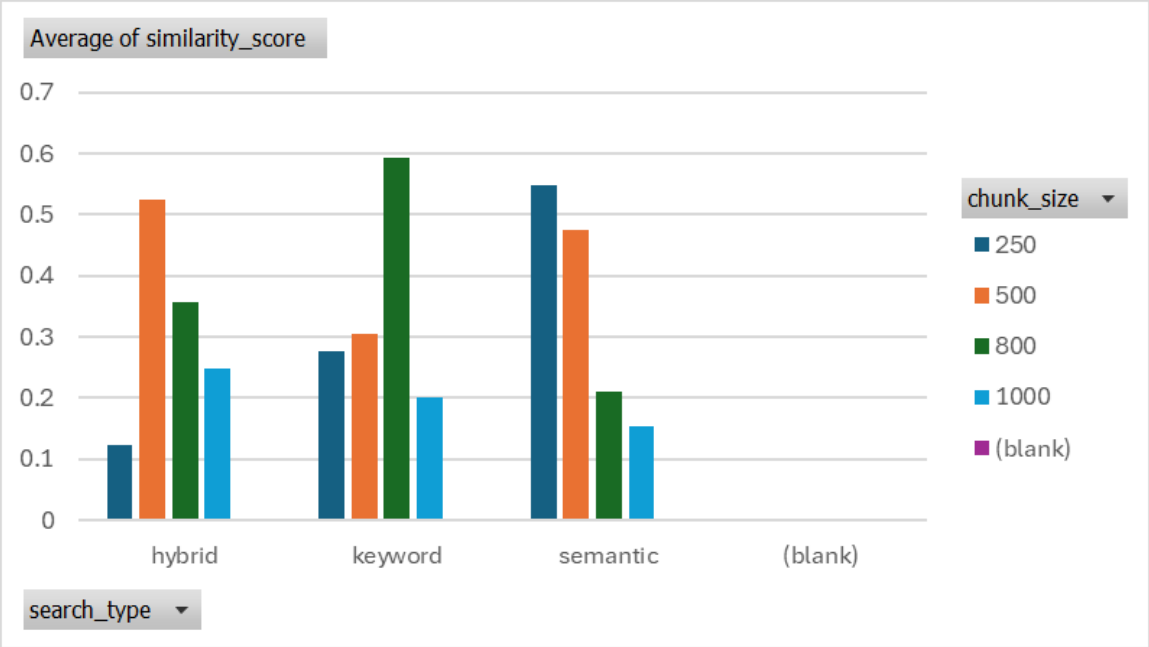


Figure 1-Bar Plot (Chunking Strategy for each Retrieval Type)

As shown in the bar chart above for *rag_grid_1A.csv*, semantic retrieval generally produced higher average relevancy scores than hybrid search. However, both retrieval methods responded similarly to changes in chunk size.

Our findings covered the following points:

- Smaller Chunks (250–500 tokens) with moderate overlap (100) yielded better retrieval fidelity, particularly with semantic search. **Larger chunks** may contain more information, reducing the number of retrievals but risk diluting relevance
- Keyword-based retrieval generally underperformed, suggesting that semantic embeddings handle nuanced queries more effectively.
- Larger chunks (750–1000) led to decreased performance, likely due to diluted relevance, especially when using hybrid or keyword searches.

- Overlaps of 100–200 tokens helped maintain answer context but increased redundancy in retrieval time.

Our Best Results were achieved at the following chunking strategy:

config	chunk_s	chunk_c	search_	top_k	semantic_v	keyword	similarity_score
chunk500_overlap100_semantic_k3	500	100	semantic	3	0.5	0.5	0.9453
chunk500_overlap100_hybrid_k4	500	100	hybrid	4	0.5	0.5	0.9179
chunk500_overlap100_keyword_k3	500	100	keyword	3	0.5	0.5	1
chunk500_overlap200_hybrid_k4	500	200	hybrid	4	0.5	0.5	0.9179
chunk800_overlap100_keyword_k3	800	100	keyword	3	0.5	0.5	0.9825
chunk800_overlap100_keyword_k5	800	100	keyword	5	0.5	0.5	0.9202
chunk800_overlap200_hybrid_k3	800	200	hybrid	3	0.5	0.5	1

Figure 2- rag_grid_1A.csv

config	chunk_s	chunk_c	search_	top_k	semanti	keywor	similarit	score
chunk250_overlap100_hybrid_k3	250	100	hybrid	3	0.5	0.5	1	
chunk250_overlap200_hybrid_k3	250	200	hybrid	3	0.5	0.5	0.9245	
chunk500_overlap100_keyword_k3	500	100	keyword	3	0.5	0.5	1	
chunk500_overlap100_keyword_k4	500	100	keyword	4	0.5	0.5	1	
chunk800_overlap100_keyword_k5	800	100	keyword	5	0.5	0.5	1	
chunk1000_overlap100_hybrid_k4	1000	100	hybrid	4	0.5	0.5	1	
chunk1000_overlap200_keyword_k4	1000	200	keyword	4	0.5	0.5	1	

Figure 3-rag_grid_1.csv

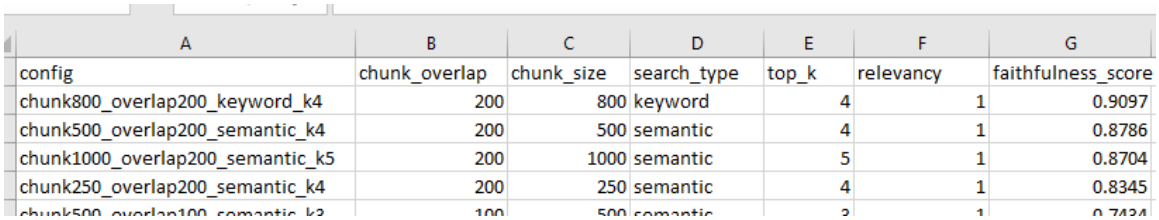
	B	C	D	E	F	G
	config	chunk_siz	chunk_overlap	top_k	search_type	Relevancy
	chunk800_overlap200_hybrid_k4	800	200	4	hybrid	1
	chunk800_overlap200_semantic_k5	800	200	5	semantic	0.9751
	chunk1000_overlap200_keyword_k5	1000	200	5	keyword	0.9214
	chunk250_overlap200_semantic_k5	250	200	5	semantic	0.9153
	chunk1000_overlap200_keyword_k4	1000	200	4	keyword	0.8972
	chunk250_overlap100_keyword_k5	250	100	5	keyword	0.8922
	chunk250_overlap200_keyword_k4	250	200	4	keyword	0.8883
	chunk800_overlap200_semantic_k4	800	200	4	semantic	0.8662
	chunk500_overlap100_keyword_k3	500	100	3	keyword	0.8627

Figure 4-grid2.csv

4. Performance Metrics

Faithfulness Score

In our evaluation grid, rag_grid_3 includes a faithfulness_score. Models evaluated with this metric provide a deeper insight into answer reliability. This can influence selection even if the system performs well in latency or similarity. Some strategies that resulted in a high relevancy score along with a decent faithfulness score are mentioned below.



	A	B	C	D	E	F	G
1	config	chunk_overlap	chunk_size	search_type	top_k	relevancy	faithfulness_score
2	chunk800_overlap200_keyword_k4	200	800	keyword	4	1	0.9097
3	chunk500_overlap200_semantic_k4	200	500	semantic	4	1	0.8786
4	chunk1000_overlap200_semantic_k5	200	1000	semantic	5	1	0.8704
5	chunk250_overlap200_semantic_k4	200	250	semantic	4	1	0.8345
6	chunk500_overlap100_semantic_k3	100	500	semantic	3	1	0.7434

Figure 5-grid3.csv

Avg Faithfulness based on LLM + chunking + search retrieval strategies is given below:

LLM	Chunk Size	Overlap	Search Type	Avg. Faithfulness
microsoft/phi-2	250	100	hybrid	0.83
microsoft/phi-2	250	100	semantic	0.74
microsoft/phi-2	250	100	keyword	0.72

Table 3- Average Faithfulness Score

Relevancy Score

A cosine-based Relevancy score between the original and regenerated question was used. RAGAS does not produce a faithfulness score, thus, to cater to this we give our generated response back to our LLM, generate relevant questions that could have possibly answered this response. Then we compute the relevancy of these questions with our original question, this is our computed Relevancy score.

We have summarized the numerical measures of these scores:

Metric	rag_grid_1.csv	rag_grid_1A.csv	rag_grid_2.csv	rag_grid_3.csv
Count of Combinations	72	72	72	72
Mean	0.393	0.334	0.6184	0.6760
Median	0.283	0.266	0.663	0.824
Minimum	-0.0918	-0.047	0.4174	0.0199
Maximum	1.0	1.0	1.0	1.0
Count of 1-maximum	6	2	1	9
Standard Deviation	0.337	0.300	0.228	0.3078
Range	1.0918	1.047	0.5826	0.9801
Variance	0.11357	0.09000	0.052166	0.0947

Table 4- Numerical Measures (CSV wise)

We tested our RAG Q/A system across 4 kinds of hyper parameters, and stored our evaluations in 4 CSVs. While *rag_grid_1.csv* captured maximum performance potential (seen through a higher average Relevancy score), consistency and stability (smaller variation and standard deviation) were seen in *rag_grid_1A.csv*.

Relevancy Score by Retrieval Type and Top-K

Retrieval Type	Top-K	Average Relevancy Score
Keyword	3	0.564
Keyword	4	0.191
Keyword	5	0.275
Hybrid	3	0.426
Hybrid	4	0.351
Hybrid	5	0.162
Semantic	3	0.406
Semantic	4	0.347
Semantic	5	0.287

Table 5-top_k and Retrieval Type Summary (grid 1 and 1A)

To summarize the findings from the table above,

- Keyword (Top-K=3) performed best overall with a score of 0.564, indicating strong precision in top results.
- Hybrid retrieval (Top-K=3) also showed high effectiveness, outperforming semantic search at the same level.
- A drop in performance was observed across all types as Top-K increased, suggesting that higher retrieval counts introduce noise, which disturbed the Relevancy score.

This is also, contained in the bar chart below for *rag_grid_1A.csv*

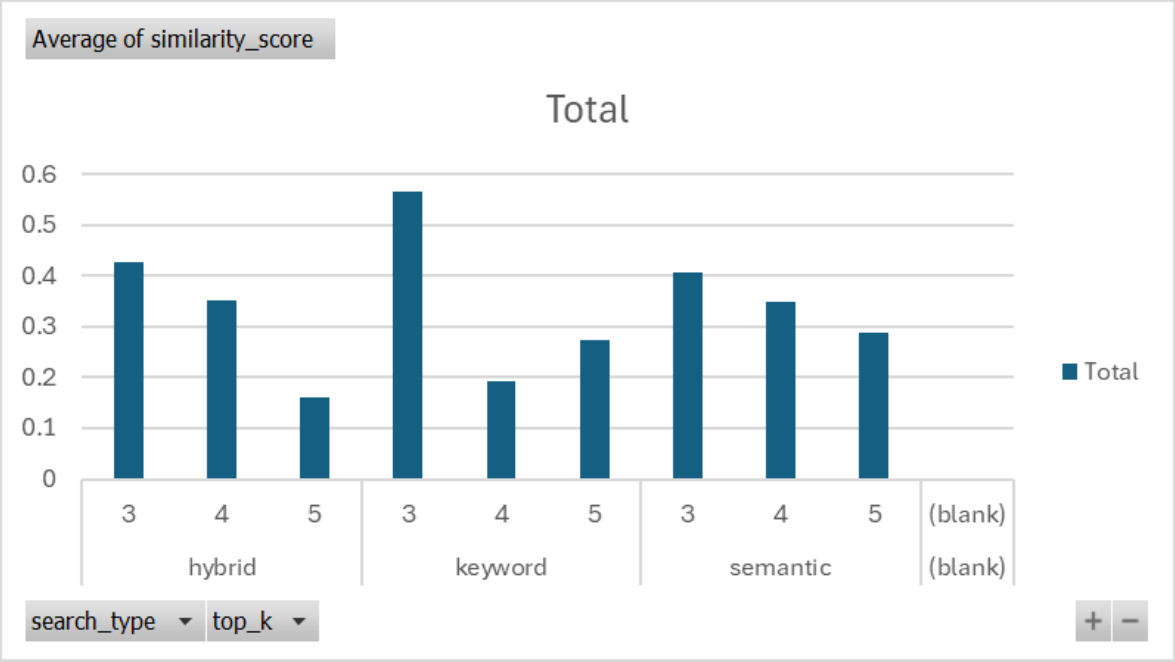


Figure 6-Bar Plot for Table 5

For rag_grid_2.csv and rag_grid_3.csv we observed,

Retrieval Type	Top-K	Avg. Similarity Score
hybrid	5	0.625
keyword	4	0.673
semantic	4	0.511

Table 6- top_k & Retrieval Type Summary (grid 2 and 3)

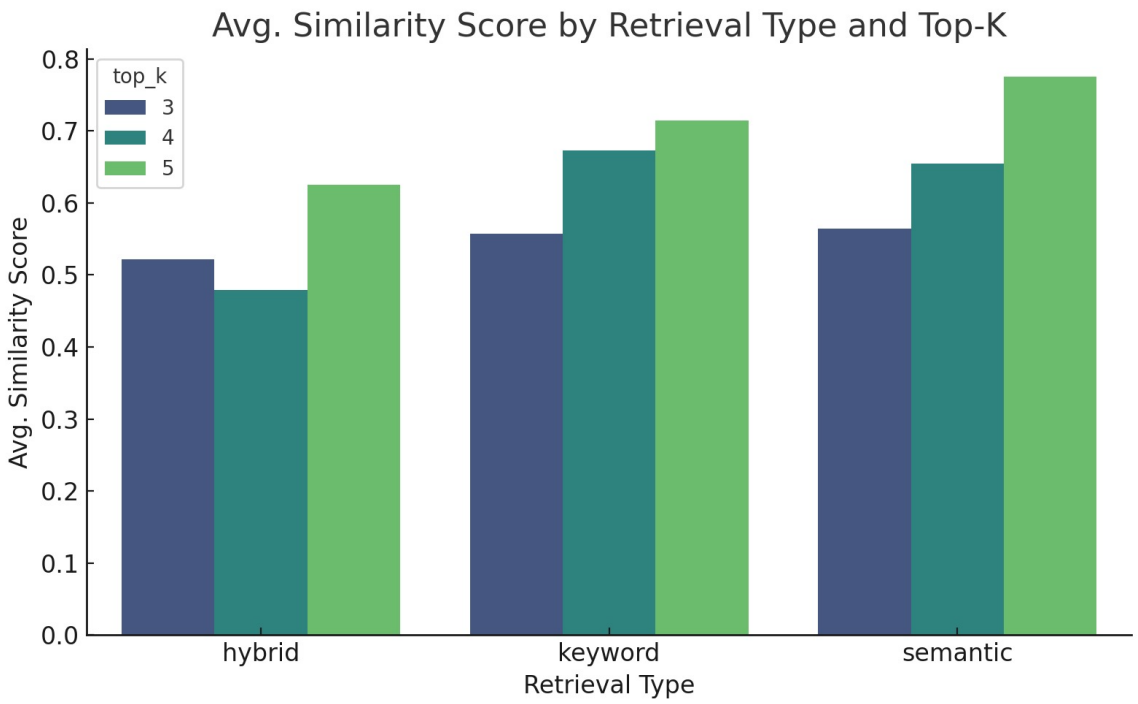


Figure 7- Bar plot for table 6

It can be concluded that: Keyword search with top_k=4 outperforms others in average similarity, indicating more relevant retrievals.

Hybrid retrieval at top_k=5 also performs well, showing the benefit of combining semantic + keyword.

Semantic retrieval at low top_k tends to underperform, likely due to its reliance on embedding similarity, which may lack precision at small contexts

A model wise Relevancy summary for rag_grid_2 and rag_grid_3 is given below:

LLM	**Chunk Size**	**Overlap**	**Retrieval**	**Avg. Similarity**
TinyLlama/TinyLlama-1.1B-Chat-v1.0	250	200	keyword	**0.751**
TinyLlama/TinyLlama-1.1B-Chat-v1.0	250	200	hybrid	0.700
TinyLlama/TinyLlama-1.1B-Chat-v1.0	250	100	keyword	0.595

microsoft/phi-2	250	100	hybrid	0.582

Table 7- Model wise Relevancy Score Summary

It is observed from the above table that:

- TinyLlama with 250/200 chunking and keyword retrieval gives the best relevancy performance overall.
- Reducing overlap to 100 slightly drops similarity, showing the benefit of higher overlap.

Latency and System Efficiency Metrics

Metric	Mean (s)	Std Dev (s)	Min (s)	Max (s)
Retrieval Time	6.76	0.39	6.17	7.83
Generation Time	1.75	0.29	0.48	2.01
Total Time	8.51	0.49	6.87	9.65

Table 8- Summary of Latency (rag1.csv and rag1A.csv)

To summarize the results from this table,

- Retrieval Time dominated the total time, constituting approximately 79% of total system latency, indicating it's the primary bottleneck.
- The generation process by the LLM is relatively stable and efficient, showing a low variance.

There could, however be a potential optimization and reasons for latency, that we discovered:

  - Retrieval efficiency could be significantly improved by integrating vector indexing,using approximate nearest neighbor (ANN) techniques.
  - Hybrid search methods may contribute to increased latency due to the computational overhead of chunk re-ranking or merging strategies.
  - Keyword searches, using the exact token comparison also contributed to a higher total time.

Impact of Retrieval type on Total time

This line chart summarizes total time variation across different search types for *rag_grid_1.csv.*

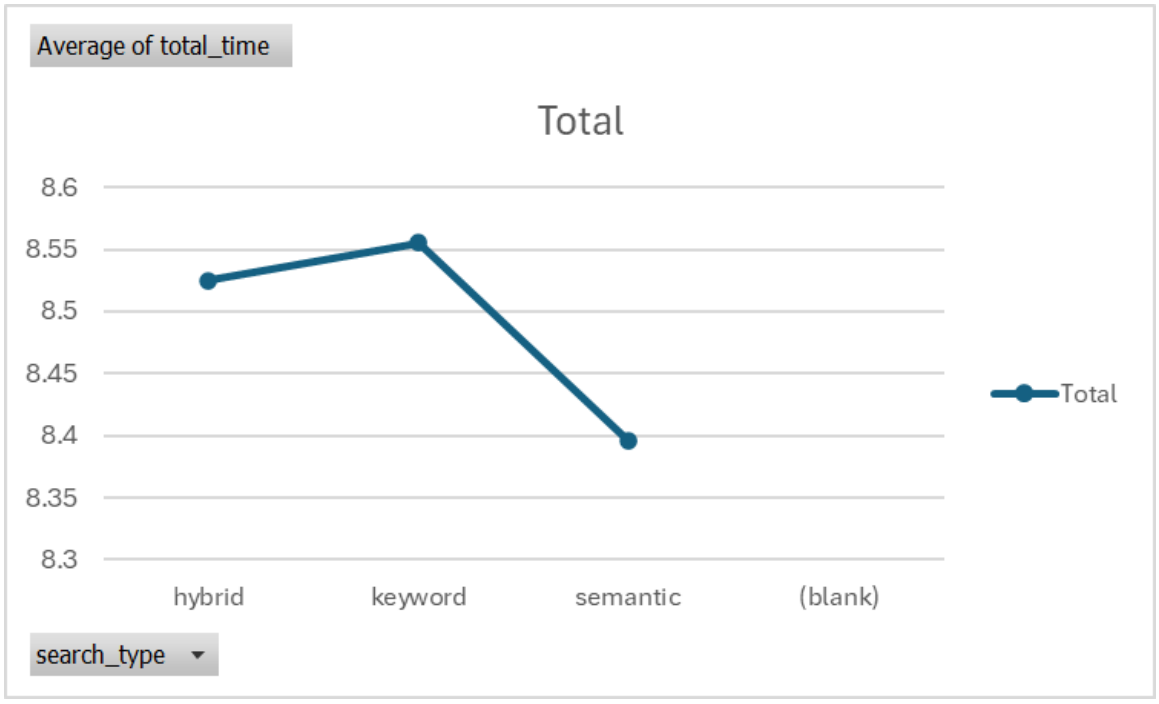


Figure 8- Line plot showing Average time for each Retrieval Type (grid 1 and 1A)

For grid_3 and grid_2:

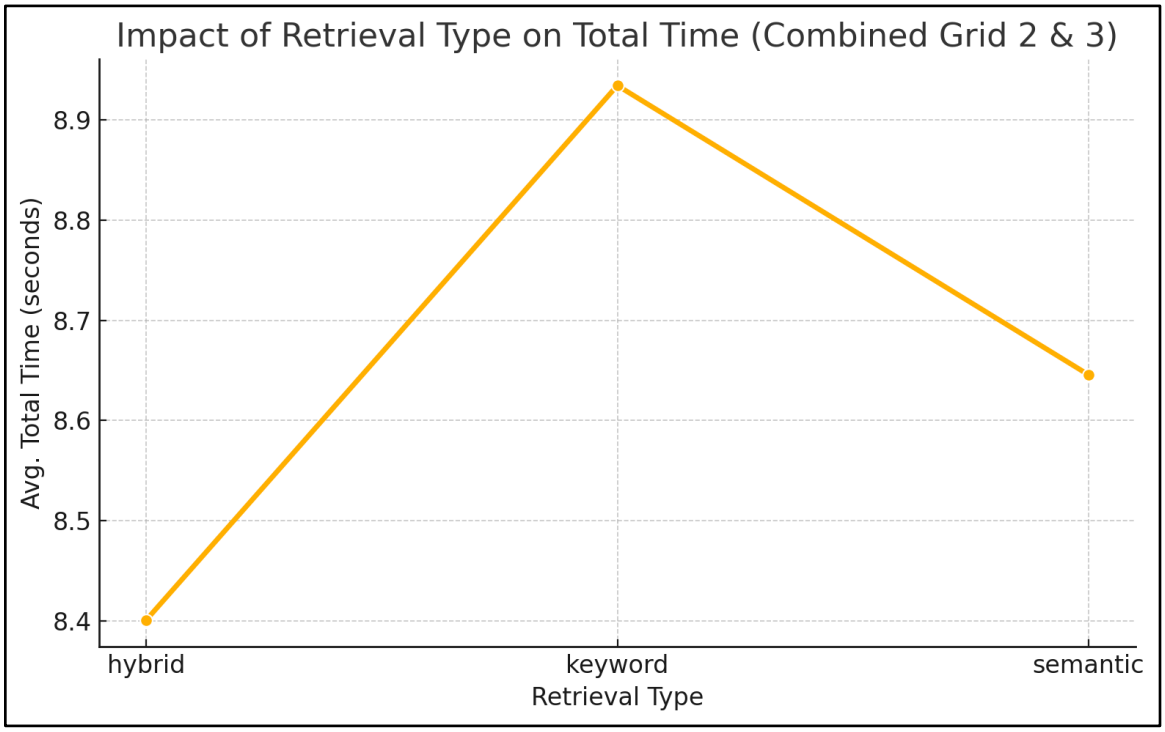


Figure 9- Time vs Retrieval Type (grid 2 and 3)

Retrieval Type	**Avg. Total Time (s)**
Hybrid	8.40
Keyword	8.93
Semantic	8.65

Table 9-Average total times (grid 2.csv and grid3.csv)

- To optimize the RAG, we infer that: Hybrid retrieval is the most time-efficient method on average.
- Keyword retrieval, while high-performing in relevancy, has the highest latency overall.
- Semantic retrieval sits in the middle in terms of latency.

5. Best Model Selection

Based on faithfulness, latency, and relevancy, we can recommend two "best" configurations, depending on your system's priority:

Priority	Model	Retrieval	Chunking	Strength
Accuracy	microsoft/phi-2	hybrid	250 / 100	High faithfulness (0.83)
Relevancy / Speed	TinyLlama	keyword	250 / 200	Top relevancy (0.75) & low latency

Table 10-Best Model Selection

6. Reproducibility

We came up with a Python code that implements a comprehensive Retrieval-Augmented Generation (RAG) QA system using a custom pipeline that integrates document ingestion, chunking, semantic and keyword search, language model response generation, and performance evaluation. The documents (e.g., lecture PDFs) are first loaded and split into overlapping chunks using LangChain's text splitters. Two retrieval methods are supported: semantic search via FAISS (using HuggingFace embeddings) and lexical search via BM25. A hybrid retrieval method combines both using weighted scores.

For response generation, HuggingFace's transformer-based models (like Mistral or LLaMA) are loaded either through a local pipeline or HuggingFace Hub. The RAG pipeline uses the top-k retrieved document chunks to construct a context-aware prompt, which is then passed to the LLM to generate an answer.

The script also includes two main evaluators. RAGEvaluator leverages the RAGAS evaluation framework with metrics like faithfulness, answer relevancy, context recall, precision, and optionally answer correctness (when gold answers are provided). Visualization tools help inspect metric averages and retrieval stats, while a recommendation module suggests optimizations based on weak metric scores. An alternative evaluator, SimpleEvaluator, uses question regeneration and cosine similarity between original and generated questions as a proxy for evaluating faithfulness and semantic alignment.

Finally, the grid_search function systematically explores different combinations of chunk sizes, overlap values, retrieval types (semantic, keyword, hybrid), and top-k values. It executes the RAG pipeline for each configuration, logs response generation and evaluation metrics (e.g., faithfulness score, question similarity), and records everything in a CSV file. This enables fine-tuning the RAG system to achieve optimal performance.

Reproducing this RAG-based QA system is straightforward thanks to its modular structure and extensive in-line documentation. All required Python packages are explicitly installed at the beginning using pip, ensuring consistency across environments. The code is designed to run smoothly on platforms like Kaggle, Google Colab, or any GPU-enabled machine with HuggingFace access. Constants like chunk size, overlap, and model names are easily configurable, enabling users to tailor the system to their own document sets. Moreover, the modular class-based design (RAGPipeline, RAGEvaluator, SimpleEvaluator) allows researchers and developers to plug in different datasets or LLMs with minimal code changes. Built-in functions for evaluation, visualization, and automated grid search simplify experimentation, making this system ideal for both beginners exploring RAG and advanced users tuning performance on custom corpora. With a Hugging Face token and access to relevant documents, anyone can reproduce the results and extend the framework for academic, enterprise, or domain-specific applications.

7. Summary of Code

The system implements a full Retrieval-Augmented Generation (RAG) pipeline that supports document ingestion, vector and keyword retrieval, LLM-based answering, and automatic evaluation.

1. Imports and Setup

The code installs and imports necessary libraries including LangChain, HuggingFace, FAISS, BM25, PyPDF, etc. It also handles authentication with HuggingFace using a login token.

2. Constants Definition

Constants like chunk size, embedding model, LLM model, and document directory are defined for flexible tuning.

3. Document Processing Functions

- `load_documents`: Loads PDF documents using LangChain's DirectoryLoader.
- `chunk_documents`: Splits large documents into overlapping chunks using RecursiveCharacterTextSplitter.

4. Vector Store and Indexing

- `create_vector_store`: Converts chunks into vector embeddings and stores them using FAISS.
- `create_bm25_index`: Creates a BM25 keyword index from document chunks for keyword-based retrieval.

5. Retrieval Methods

- `semantic_search`: Uses FAISS similarity search.
- `keyword_search`: Uses BM25 keyword-based scoring.
- `hybrid_search`: Combines semantic and keyword scores using weighted fusion.

6. Language Model Initialization

- `initialize_llm`: Loads a HuggingFace text generation model and tokenizer, wrapped into a pipeline.

7. Prompt and Response Generation

- `format_rag_prompt`: Formats a structured prompt by combining retrieved context and question.
- `generate_response`: Uses the model to generate an answer based on the formatted prompt.

8. Evaluation and Wrappers

- `RAGEvaluator`: Runs RAGAS-based evaluations using metrics like Faithfulness and Relevance.
- `SimpleEvaluator`: Custom evaluator that generates a reverse question and computes cosine similarity.
- `HuggingFaceLLMWrapper` and `HuggingFaceLLM`: Used to adapt HuggingFace pipelines for use in evaluation libraries.

9. Core RAGPipeline Class

Main class that combines all pipeline steps:
- `load_and_process_documents`
- `initialize_retrieval`
- `initialize_llm`
- `query`: Executes full RAG pipeline for a question.
- `experiment` and `grid_search`: Run various configurations for testing retrieval and generation performance.

10. Execution & Usage

The final part runs the pipeline step-by-step:
- Initialize embeddings and LLM
- Load documents and create indexes
- Run a sample query and evaluate it
- Conduct a grid search across different chunk sizes, search types, and k-values.

Table of Figures & Tables