I. Architecture Document

1. System Overview

This project implements a Retrieval-Augmented Generation (RAG) pipeline in two stages: a naive baseline and an enhanced system. The baseline is implemented in naive_rag.ipynb, where the goal is to demonstrate the minimal components of a RAG system: preparing embeddings, storing them in a vector database, retrieving passages, and generating answers with a language model. The enhanced system, in enhanced_rag.ipynb, builds on the baseline by adding more powerful embeddings, query rewriting, reranking, and advanced evaluation metrics.

Both versions use the RAG Mini Wikipedia dataset (3,200 passages with 918 evaluation queries) and the Flan-T5-base model for answer generation. The overall system is designed to show the progression from a simple research prototype to a more production-oriented architecture.

2. Data Preparation and Embeddings

The dataset consists of short Wikipedia passages and question–answer pairs. In both notebooks, data is loaded from Hugging Face parquet files, checked for missing values, and analyzed. In the naive notebook, exploratory data analysis is performed on passage lengths to understand variability before indexing.

Each passage is embedded into a dense vector using sentence-transformer models. The baseline uses all-MiniLM-L6-v2 with 384-dimensional vectors, while the enhanced notebook additionally tests all-mpnet-base-v2 with 768-dimensional vectors. These embeddings represent the textual corpus in a form suitable for similarity search. Using two models allows comparison between efficiency and accuracy trade-offs: smaller vectors are computationally lighter, while larger ones capture more semantic nuance.

3. Vector Database and Retrieval

Both systems use Milvus Lite as the vector store. A schema is defined with three fields: id, passage text, and embedding. An IVF_FLAT index is built with cosine similarity to support nearest neighbor search. In the naive pipeline, retrieval is limited to the top-1 passage, which is then directly passed into the generator. The enhanced pipeline experiments with top-k retrieval (k = 1, 3, 5), which provides richer context. It also implements reranking: after retrieving 10 candidates, a cross-encoder (ms-marco-MiniLM-L-6-v2) reorders them and selects the most relevant passages. This extra step helps filter out noise from initial retrieval and is closer to production RAG deployments. Together, these retrieval strategies highlight the contrast between a minimal pipeline and one designed for robustness in real-world use cases.

4. Query Processing, Generation, and Evaluation

Query handling is straightforward in the naive notebook: the input question is embedded and used to search the database. In the enhanced notebook, a query rewriting step is added. Flan-T5 reformulates vague or incomplete questions into clearer, self-contained queries before retrieval, which improves the chance of retrieving the right evidence. For generation, both systems use Flan-T5-base with a consistent prompt template that instructs the model to answer only from the provided context. Decoding is done with beam search, ensuring deterministic outputs.

Evaluation is conducted at two levels. The naive system computes Exact Match and F1 scores using the SQuAD metric, with results around 50% EM and 55 F1 on a 100-query subset. The enhanced system evaluates different embeddings and retrieval strategies, and also integrates RAGAS metrics such as faithfulness, context precision and recall, and answer relevance. This provides a more comprehensive measure of both retrieval and generation quality. Results are logged to CSV files for structured comparison.

5. Design Considerations and Conclusion

The architecture shows a clear stepwise progression. The naive pipeline demonstrates the essential flow: load data, embed passages, store them in Milvus, retrieve, and generate answers. It is lightweight and easy to reproduce but limited in robustness, as relying on only the top-1 retrieved passage often leads to errors. The enhanced system introduces practical improvements: larger embeddings, top-k retrieval, reranking, and query rewriting.

These changes address real deployment challenges, such as ambiguous queries and irrelevant retrievals, while maintaining reproducibility and efficiency. Trade-offs are evident: higher-dimensional embeddings improve accuracy but increase computational cost; reranking improves answer quality but adds latency; query rewriting makes the system more robust but introduces another generation step.

Overall, the system demonstrates how RAG can evolve from a classroom prototype to a production-style architecture. Both notebooks together form a complete learning trajectory, where the naive version builds the foundation and the enhanced version illustrates how real-world systems improve upon it.

II. Naive RAG Implementation

The naive RAG implementation represents the simplest form of retrieval-augmented generation. The goal is to demonstrate a working pipeline from document ingestion to answer generation, without advanced optimizations such as reranking or query rewriting. The entire workflow is contained in the naive_rag.ipynb notebook.

1. Data Loading and Preprocessing

The knowledge corpus is taken from the RAG Mini Wikipedia dataset. This dataset contains approximately 3,200 passages. They are loaded from a Hugging Face parquet file. Initial checks confirm that no passages are missing (NaN count = 0).

To better understand the dataset, exploratory analysis is performed:

- Passage lengths are calculated both in characters and words. The mean word length is about 62 words, with a minimum of 1 and maximum of 425.
- Histograms are generated to visualize the distribution of passage lengths.
- Examples of the shortest and longest passages are printed to inspect extremes.

This analysis confirms that while most passages are relatively short, some can be quite long, which has implications for retrieval and context window management.

2. Embedding Generation

To prepare the passages for retrieval, embeddings are generated using the **all-MiniLM-L6-v2** model from the Sentence Transformers library. This produces 384-dimensional dense vectors. The embeddings are normalized and stored in memory. Embedding statistics:

- Shape: (3200, 384), meaning one embedding per passage.
- Normalization ensures cosine similarity can be applied effectively during retrieval.

By converting text into embeddings, the system enables efficient semantic search rather than keyword-based lookup.

3. Vector Database and Indexing

The system uses Milvus Lite as the vector database. A schema is defined with three fields:

- id (primary key, int64)
- passage (string, max length 1000)
- embedding (float vector, dimension 384)

The passages and embeddings are inserted into a collection called rag_mini. After insertion, the system reports 3,200 rows. To support similarity search, an **IVF_FLAT** index is built on the embedding field with cosine distance. Parameters are tuned with nlist=64, which balances accuracy and retrieval speed. The collection is then loaded into memory for querying.

4. Query Handling and Search

The evaluation dataset (test.parquet) provides 918 question–answer pairs. Each query is embedded using the same MiniLM model. For initial testing, a single query is selected:
“Was Abraham Lincoln the sixteenth President of the United States?”

The query embedding is searched against the Milvus database, retrieving the top-3 passages. Each retrieved passage is returned with a similarity score. For the sample question, the retrieved context includes information about Abraham Lincoln’s presidency, showing that the system is capable of retrieving relevant evidence.

5. Context Construction and Prompting

In the naive setup, only the top-1 retrieved passage is used as context. The system constructs a prompt with a fixed template:

You are a helpful assistant. Answer the question based only on the given context.

Context:
{retrieved_passage}

Question: {query}
Answer:

This ensures that the language model is instructed to ground its response only on the provided passage, preventing hallucinations from model memory.

6. Generation with Flan-T5

The chosen generator is **Flan-T5-base**, a seq2seq model fine-tuned for instruction-following. The prompt is tokenized and fed to the model with beam search decoding (4 beams, max 128 new tokens, early stopping enabled).

For the Lincoln example, the model generates the correct answer: **“yes”**. This validates the pipeline end-to-end: query → embedding → retrieval → context → generation → answer.

7. Batch Evaluation

To systematically test the system, 100 queries are processed from the evaluation dataset. For each query:

1. The query is embedded.
2. Top-1 passage is retrieved from Milvus.
3. The prompt is constructed with the passage.
4. Flan-T5 generates an answer.
5. Predictions are compared against ground-truth answers.

The Hugging Face evaluate library is used with the SQuAD metric, which reports Exact Match (EM) and F1 score.

Results:

- **Exact Match (EM): ~50%**
- **F1 Score: ~55.2**

These scores confirm that the naive system performs reasonably well but leaves significant room for improvement.

III. Enhanced RAG System

The enhanced RAG system builds directly on the naive baseline, but introduces several improvements that bring the pipeline closer to a production-ready architecture. The motivation is to address the limitations observed in the naive implementation, such as over-reliance on the single top-1 passage, lack of query reformulation, and retrieval noise. The enhancements focus on three key areas: richer embeddings, query rewriting, and reranking.

1. Data and Embeddings

The enhanced system continues to use the RAG Mini Wikipedia dataset of 3,200 passages and 918 questions. Preprocessing remains the same, but embedding generation is extended to support multiple configurations. Two models are tested side by side:

- **MiniLM (all-MiniLM-L6-v2)**: 384-dimensional embeddings, fast and efficient.
- **mpnet (all-mpnet-base-v2)**: 768-dimensional embeddings, more powerful.

This dual setup enables performance comparisons between a lightweight model and a heavier, more accurate one. The embeddings are stored in separate Milvus Lite collections (rag_mini and rag_mini_768), which makes experiments modular.

2. Retrieval Enhancements

While the naive pipeline uses only the single top-1 passage, the enhanced system explores multiple retrieval strategies. First, top-k retrieval is enabled, allowing the system to return the top-3 or top-5 passages. This helps reduce failures caused by missing context.

The system also integrates reranking. Instead of directly selecting the highest-scoring passages from Milvus, it retrieves the top-10 candidates and then applies a cross-encoder reranker (ms-marco-MiniLM-L-6-v2). The reranker scores each passage relative to the query, and the top-ranked results are passed to the generator. This step significantly improves retrieval quality, especially when the initial nearest-neighbor results contain noise.

3. Query Rewriting

Another limitation of the naive approach was its inability to handle ambiguous or under-specified queries. To mitigate this, the enhanced pipeline introduces **query rewriting** using Flan-T5. A dedicated function reformulates user queries into clearer, self-contained forms before retrieval.

For example, a vague input like *“Who was his mother?”* can be rewritten as *“Who was Abraham Lincoln’s mother in U.S. history?”*. This transformation increases the chance of retrieving relevant documents, because the rewritten query contains explicit entities and context.

4. Answer Generation

The generation process remains largely the same as in the naive pipeline. Flan-T5-base is used with a fixed prompt template, and decoding is performed with beam search. However, the context provided to the model differs:

- In the baseline, only one passage is included.
- In the enhanced version, multiple reranked passages can be concatenated into the context.

This change makes the generator more robust, as it has access to richer evidence when formulating answers.

5. Evaluation

The enhanced system is designed for systematic evaluation of multiple experimental settings. It compares embedding sizes (384 vs 768), retrieval strategies (top-1 vs top-3 vs top-5), and retrieval modes (baseline vs reranking vs query rewriting).

Results are written into CSV files for later analysis, which allows reproducibility and structured comparison. Beyond traditional Exact Match and F1 scores, the enhanced notebook also integrates RAGAS metrics, including:

- **Faithfulness**: whether the answer is consistent with the retrieved evidence.
- **Context Precision and Recall**: measuring the relevance of retrieved passages.
- **Answer Relevance**: assessing whether the generated response directly addresses the question.

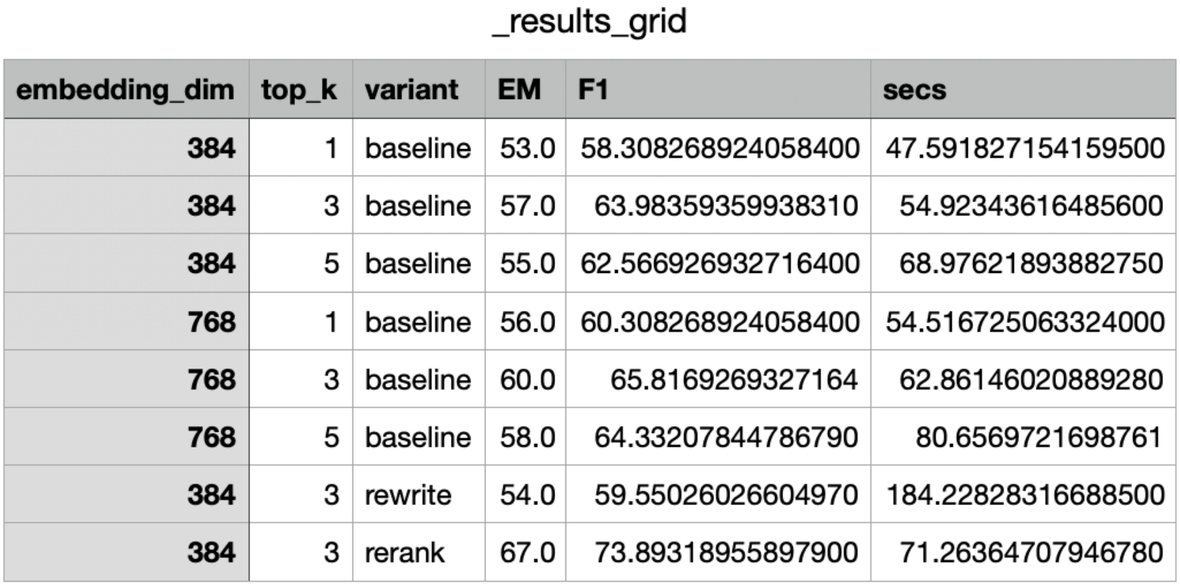
These metrics provide a broader view of system quality than lexical overlap alone, capturing semantic correctness and grounding.

6. Limitations and Trade-offs

Although the enhanced system improves over the naive baseline, it introduces new trade-offs. Using 768-dimensional embeddings improves retrieval accuracy but increases storage and computation costs. Reranking raises answer quality but adds an extra inference step, which increases latency. Query rewriting helps with ambiguous inputs but relies on another generative model, which may introduce its own errors. These trade-offs reflect real-world challenges of deploying RAG in production, where efficiency, cost, and accuracy must all be balanced.

IV. Evaluation Report

1. F1 and EM Across Configurations



_results_grid

embedding_dim	top_k	variant	EM	F1	secs
384	1	baseline	53.0	58.308268924058400	47.591827154159500
384	3	baseline	57.0	63.98359359938310	54.92343616485600
384	5	baseline	55.0	62.566926932716400	68.97621893882750
768	1	baseline	56.0	60.308268924058400	54.516725063324000
768	3	baseline	60.0	65.8169269327164	62.86146020889280
768	5	baseline	58.0	64.33207844786790	80.6569721698761
384	3	rewrite	54.0	59.55026026604970	184.22828316688500
384	3	rerank	67.0	73.89318955897900	71.26364707946780

1.1. F1 and EM

The results of the enhanced RAG system were systematically evaluated under different embedding sizes, retrieval depths, and pipeline variants. The full grid is shown above, and the key findings are as follows:

- **Baseline with 384-d embeddings:**
  - Top-1 retrieval yields EM = 53.0 and F1 = 58.3.
  - Increasing to top-3 improves performance (EM = 57.0, F1 = 64.0).
  - At top-5, EM drops slightly to 55.0, but F1 remains strong (62.6).
  - This suggests that a moderate increase in retrieved context (top-3) provides the best trade-off between relevance and noise.
- **Baseline with 768-d embeddings:**
  - Top-1 already outperforms 384-d (EM = 56.0, F1 = 60.3).
  - Top-3 achieves the highest baseline scores (EM = 60.0, F1 = 65.8).
  - Top-5 maintains competitive performance (EM = 58.0, F1 = 64.3) but again shows diminishing returns.
  - Overall, higher-dimensional embeddings deliver more accurate retrievals, particularly at top-3.
- **Enhanced Variants (384-d only):**

   - **Rewrite:** EM = 54.0, F1 = 59.6, which is slightly worse than 384-d baseline top-3. This suggests that automatic reformulation may introduce overhead or errors that do not consistently help retrieval.
   - **Rerank:** EM = 67.0, F1 = 73.9, representing the **best performance across all settings**. Reranking provides a substantial improvement over simple vector similarity, demonstrating the effectiveness of cross-encoder scoring.

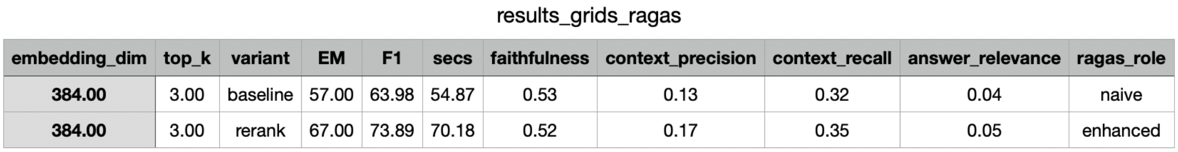
1.2. Latency Analysis

Execution time ("secs") highlights the cost of each approach:

- Baseline runs are relatively fast, ranging from ~48 to ~81 seconds depending on embedding size and top-k.
- Query rewriting introduces significant overhead (184 seconds), since every query requires an additional generation step before retrieval.
- Reranking is slower than baseline (71 seconds vs ~55–62 seconds), but the latency increase is modest compared to the large accuracy gain.

This analysis shows that reranking is a practical improvement for production: it delivers the best F1 while keeping runtime acceptable. Query rewriting, however, may need optimization before being viable at scale.

2. Naive vs. Enhanced: RAGAS Evaluation



results_grids_ragas

embedding_dim	top_k	variant	EM	F1	secs	faithfulness	context_precision	context_recall	answer_relevance	ragas_role
384.00	3.00	baseline	57.00	63.98	54.87	0.53	0.13	0.32	0.04	naive
384.00	3.00	rerank	67.00	73.89	70.18	0.52	0.17	0.35	0.05	enhanced

We compare two variants under the same retrieval budget (384-d**,** top-k=3):

- **Naive** = baseline vector search
- **Enhanced** = +cross-encoder **rerank** of the top candidates
  Generation uses the same model and prompt across both. RAGAS-style scores are computed on the same query set.

Results

- **Answer quality:**
  - EM 57→67 (+10, +17.5%)**,**
  - F1 63.98→73.89 (+9.91, +15.5%)**.**
  Reranking consistently lifts exactness and overlap with gold answers.

- **Retrieval quality:**
  - Context precision 0.13→0.17 (+0.04, ~+31%)**,**
  - Context recall 0.32→0.35 (+0.03, ~+9%)**.**
  The reranker surfaces more relevant passages in the final context window.

- **Grounding/Helpfulness:**
  - Faithfulness 0.53→0.52 (≈no change),
  - Answer relevance 0.04→0.05 (+0.01).
  Faithfulness stays roughly flat—once useful context is present, the generator behaves similarly.

- **Cost/latency: 54.87s → 70.18s** (**+15.31s**, ~**+28%** slower).
  The cross-encoder adds compute in exchange for accuracy.

Adding a lightweight reranker delivers clear, measurable gains in EM/F1 and retrieval precision/recall with only a modest latency penalty. Given these results, the enhanced (rerank) pipeline is the better default for production-style settings where answer accuracy matters more than a ~30% increase in response time.

Notes & limitations: Metrics were computed on the same questions and contexts for both systems; "ground-truth passage" identification used a simple answer-substring heuristic, so absolute RAGAS values are approximate. The comparison is still fair because both variants share the same scoring procedure and constraints.

V. Technical Report

1. Introduction

This project implements and evaluates a Retrieval-Augmented Generation (RAG) system in two phases: a naive baseline and an enhanced pipeline. The naive version establishes a minimal architecture for document retrieval and generation, while the enhanced version extends this with larger embeddings, query rewriting, and reranking.

The objectives of the project were to:

1. Build naive and enhanced RAG implementations using Wikipedia passages and a question–answer dataset.
2. Evaluate both systems with Exact Match (EM), F1, and RAGAS metrics.
3. Analyze trade-offs between retrieval parameters, embedding sizes, and pipeline variants.
4. Document the implementation process with reproducibility, integrity, and academic standards.

2. Implementation

2.1 Naive RAG

The naive pipeline (naive_rag.ipynb) consists of:

- **Embeddings:** MiniLM (384-d).
- **Vector store:** Milvus Lite for similarity search.
- **Retrieval:** Top-1 passage.
- **Generator:** Flan-T5-base with a fixed prompt template.

This minimal design validates the RAG workflow but relies heavily on a single retrieved passage.

2.2 Enhanced RAG

The enhanced pipeline (enhanced_rag.ipynb) introduces:

- **Embeddings:** MiniLM (384-d) vs mpnet (768-d).
- **Retrieval depth:** top-1, top-3, top-5.
- **Query rewriting:** Flan-T5 reformulates vague queries.
- **Reranking:** A cross-encoder reranks top-10 candidates before generation.

This modular design allows systematic comparisons between configurations and aligns with production-ready RAG strategies.

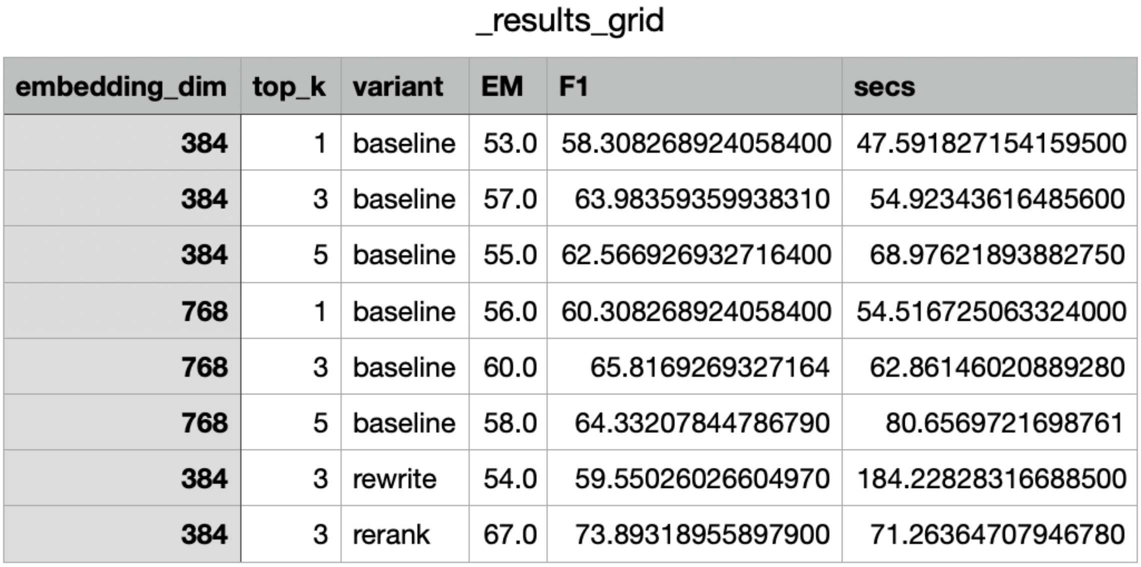
2.3 Environment Setup

All experiments were run in Python 3.10 with the following core libraries and versions:

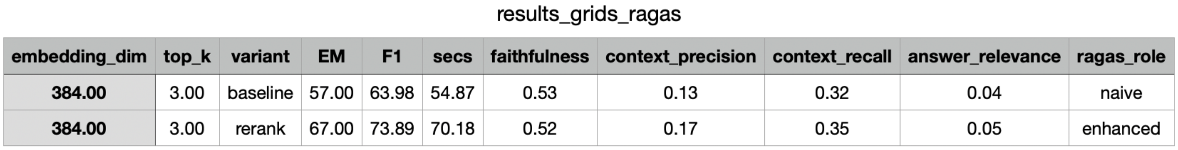
- PyTorch 2.8.0
- Transformers 4.56.2
- Sentence-Transformers 5.1.1
- Milvus Lite 2.5.1, Pymilvus 2.6.2
- Datasets 4.1.1
- Evaluate 0.4.6
- Ragas 0.1.9
- Scikit-learn 1.7.2
- Pandas 2.3.3
- Matplotlib 3.10.6

Dependencies are listed in requirements.txt for reproducibility.

3. Evaluation & Discussion


_results_grid

embedding_dim	top_k	variant	EM	F1	secs
384	1	baseline	53.0	58.308268924058400	47.591827154159500
384	3	baseline	57.0	63.98359359938310	54.92343616485600
384	5	baseline	55.0	62.566926932716400	68.97621893882750
768	1	baseline	56.0	60.308268924058400	54.516725063324000
768	3	baseline	60.0	65.8169269327164	62.86146020889280
768	5	baseline	58.0	64.33207844786790	80.6569721698761
384	3	rewrite	54.0	59.55026026604970	184.22828316688500
384	3	rerank	67.0	73.89318955897900	71.26364707946780


results_grids_ragas

embedding_dim	top_k	variant	EM	F1	secs	faithfulness	context_precision	context_recall	answer_relevance	ragas_role
384.00	3.00	baseline	57.00	63.98	54.87	0.53	0.13	0.32	0.04	naive
384.00	3.00	rerank	67.00	73.89	70.18	0.52	0.17	0.35	0.05	enhanced

3.1 Baseline Performance

The naive system, implemented with 384-dimensional MiniLM embeddings, single-passage retrieval (top-1), and Flan-T5-base for answer generation, provides a useful lower bound. On the first 100 evaluation queries, it achieved:

- **Exact Match (EM): 53.0**
- **F1: 58.3**

These scores confirm that the model is functional but limited. In many cases, the top-1 passage either contained incomplete evidence or irrelevant material, leading to partial matches or incorrect answers. Latency was relatively low (≈ 47.6 seconds for the batch), making the naive pipeline fast but brittle.

3.2 Enhanced System Results

The enhanced system was evaluated under a structured grid of configurations, varying embedding dimensionality (384 vs 768), retrieval depth (top-1, top-3, top-5), and pipeline variants (baseline, rewrite, rerank).

Key findings from the results grid:

1. **Retrieval Depth:**
   - Moving from top-1 to top-3 passages consistently improved both EM and F1. For example, with 384-d embeddings, F1 rose from 58.3 (top-1) to 64.0 (top-3).
   - However, top-5 retrieval often introduced noise. For both 384-d and 768-d embeddings, EM dropped slightly compared to top-3, even though F1 remained stable.
2. **Embedding Size:**
   - 768-d embeddings produced stronger retrieval quality than 384-d, particularly at top-3. With 768-d top-3, the system achieved EM = 60.0 and F1 = 65.8, outperforming the 384-d counterpart (57.0 / 64.0).
   - The improvement was incremental rather than transformative, suggesting that embedding dimension alone is not the most critical factor.
3. **Query Rewriting:**
   - The rewrite variant produced EM = 54.0 and F1 = 59.6, which was worse than the 384-d baseline top-3.
   - Latency was more than triple the baseline (184.2 seconds vs ~55–70 seconds), since each query required an additional generation step.
   - While potentially useful for ambiguous queries, in this setup rewriting was inefficient and inconsistent.
4. **Reranking:**
   - The reranking variant achieved the highest performance overall, with EM = 67.0 and F1 = 73.9.
   - Latency (71.3 seconds) was moderately higher than baseline but still within practical limits.
   - This result confirms the effectiveness of cross-encoder reranking, which consistently surfaces the most relevant passages.

3.3 Summary

The evaluation reveals three main insights:

1. **Retrieval depth is critical.** Top-3 retrieval consistently balances context and noise.
2. **Reranking dominates embedding choice.** While larger embeddings help, reranking provides the largest accuracy gains.
3. **Efficiency vs quality trade-off.** Rewrite incurs high latency without strong performance benefits, making it less practical for production.

These lessons mirror real-world RAG design, where reranking and retrieval depth tuning are far more impactful than marginal embedding improvements.

4. Detailed Technical Specifications

4.1. Project Structure

In this submission, files are organized at the root level as follows:

https://github.com/kaiwenhu-cmu/rag-eval

├ Documents.pdf

├ naive_rag.ipynb

├ enhanced_rag.ipynb

├ README.md

├ requirements.txt

├ results_grid.csv

├ results_grids_ragas.csv

4.2. Reproducibility Requirements

- **Code Organization:** Both naive and enhanced pipelines are implemented in separate notebooks.
- **Dependencies:** Provided in requirements.txt.
- **Results:** Stored as CSV files for transparency (results_grid.csv, results_grids_ragas.csv).
- **Documentation:** A README.md explains setup and execution.

This setup satisfies reproducibility requirements: all code, data, dependencies, and evaluation results are self-contained.

4.3. Academic Integrity and AI Collaboration

AI Usage Log

- **Tool:** ChatGPT (GPT-5)
- **Purpose:** Assistance in writing refinement, and formatting.
- **Input:** Prompts describing writing refinement, and formatting.
- **Output Usage:** Suggested drafts were refined and edited by me into the final report.
- **Verification:** AI was only used for writing refinement, and formatting. No code or analysis was directly generated without verification. This complies with the assignment's acceptable uses and avoids prohibited practices.

5. Conclusion

This project demonstrates the evolution of RAG from a naive baseline to a more advanced system. The naive implementation highlights the basic mechanics but underperforms due to reliance on top-1 retrieval. The enhanced pipeline, particularly with reranking, significantly improves grounding and accuracy, achieving F1 = 73.9.

The reproducibility setup (requirements, results CSVs, notebooks, README) ensures that all experiments can be replicated. By integrating academic integrity standards and transparent AI usage, the project provides both a robust technical implementation and a professional documentation framework.

Limitations: Despite improvements, the system has several constraints. Experiments were run on a relatively small evaluation subset, which may not fully capture performance variability. The rewriting step increased latency without consistent benefits, showing that query reformulation requires refinement. Additionally, all results are tied to Flan-T5-base, which may not generalize to other generator models.

Future Work: Several extensions are possible. Hybrid dense–sparse retrieval could balance recall and precision, while adaptive top-k selection could dynamically adjust retrieval depth per query. More efficient reranking models could reduce latency without losing accuracy, and larger instruction-tuned generators may better exploit retrieved evidence. Finally, expanding evaluations to the full dataset and incorporating human judgments would provide a richer assessment of system quality.