CSE 234 Project Report

Team 33 - Jiayin Meng & Ronit Agarwala

1. Data Creation Methodology

For generating the Q&A pairs, we used an LLM-assisted approach, followed by manual verification and filtering. We took the six categories - factual lookup, conceptual explanation, procedural, comparative, feature enumeration, and multi-file synthesis - given in the released validation Q&A set, and prompted the LLM to generate unique and new questions from the corpus.

- First, we concatenated all the .rst files in the source docs into one giant file. Each line in this newly concatenated file will have a “global” line number, indicating its number in the joined corpus. This number is appended to the very beginning of each line. During concatenation, we also maintain a “line_map”, which maps each global line number to the name of the specific .rst file it was taken from, along with its local line number within that file (global_line → (filename, local_line)).
- Next we prompted the LLM each time to take the entire concatenated corpus (along with the appended global line numbers), as well as the 45 released validation Q&A pairs, and generate completely new question-answer pairs. The LLM is instructed to generate questions of a specific category at a time. For the output it generates three keys - "question", "reference_answer", "source_evidence". “source_evidence” is a list of objects, each with one key: "global_lines": [start, end] (inclusive). This way the LLM itself gives us the line spans for the reference answer, which we can later map to the original files and local line numbers.
- To map the LLM-generated global line spans to the local files and local line numbers, we just use our previous “line_map”. Our output json thus contains the "source_evidence" key containing a list of the .rst file names and the local line spans.
- We generate a total of 42 questions - 7 for each category. We then manually go through each question-answer pair, checking to see if there are any duplicates with the released set, or with the generated questions themselves. We also check to see if the LLM-generated answers and line spans are accurate, by going through the source evidence files for each question in our output json.
- During manual evaluation we noticed that some of the LLM generated questions did belong to multiple categories (for example, many factual lookup questions were also feature enumeration), which we then added as fields in our output json as well, along with their unique question ids.
- Finally, we filter a total of 20 unique Q&A pairs for our final golden dataset.

2. Final Pipeline Architecture

Chunking Strategy:

- We implemented a custom splitter that splits documents at RST section boundaries (i.e. lines of repeated symbols like ===, —, ~~~ following with a title line) so that we could ensure each chunk corresponds to a semantically complete section with its title preserved.
- For sections that exceed 512 tokens, it falls back to use `RecursiveCharacterTextSplitter.from_tiktoken_encoder` with chunk_size = 512 and chunk_overlap = 64.

- This approach outperformed fixed-size splitting because RST documentation has a well-defined hierarchical structure. By splitting at section boundaries, we prevents cutting through mid-section content, which improves both retrieval precision and the coherence of context passed to the generator.

Embedding Model:

- We use BAAI/bge-large-en-v1.5 via HuggingFace Embeddings, running on CPU with normalized embeddings.

Indexing & Retrieval:

- FAISS
- Similarity search with k=10, retrieving the top 10 candidate chunks per query.

Reranking:

- We use cross-encoder/ms-macro-MiniLM-L-12-v2 CrossEncoder reranker, selecting top_n=1 from the 10 candidates.

Prompt Design:

- Generator: claude-sonnet-4-6 with max_completion_tokens=512.
- System Message:
  A concise instruction specifying the assistant role, context-only constraint, accuracy and completeness requirements, conciseness requirement, and a fallback response for insufficient context.
- User Message:
  "Question:{query}\n\nContext:\n{context}\n\nAnswer:"
  Placing the question before the context so the model knows what to look for before reading the retrieved context.

Context Assembly within Budget:

- The per-query context budget is 2000 tokens.
- Retrieved context is serialized and then passed through a custom function, which uses tiktoken to count tokens and truncates the tail of the retrieved context if the total tokens of fixed prompt text and the retrieved context exceeds the budget.
- With top_n=1 and chunk_size=512, a single chunk occupies at most 512 tokens, which leaves enough room for the fixed prompt.

3. Experimentation Methodology

We used RapidFire AI's RFGridSearch API to systematically compare configurations across all major RAG pipeline components. For each experiment, we defined candidate knob values using List() and RFGridSearch automatically expanded them into all combinations as distinct configs. We present the most meaningful 9 experiments with a total of 39 distinct configs. Each experiment fixed all knobs except the ones under investigation, allowing clean comparisons. The table below shows config knob details.

Run	Splitter	Chunk Size	Overlap	Embedding	Search Type	k	Reranker	top_n	Generator	Max Tokens	Prompt	Ret. Score	Gen Score
Run 1 — Baseline													
1-1	Recursive	512	32	all-MiniLM-L6-v2	similarity	5	None	–	mistral-small	512	baseline prompt	0.443	0.748
Run 2 — Reranker Test													
2-1	Recursive	512	32	all-MiniLM-L6-v2	similarity	5	CrossEncoder (L6-v2)	3	mistral-small	512	baseline prompt	0.503	0.794
2-2	Recursive	512	32	all-MiniLM-L6-v2	similarity	10	CrossEncoder (L6-v2)	3	mistral-small	512	baseline prompt	0.526	0.802
Run 3 — Fine-tuned Embedding Model Test (chunk=320, overlap=48, k=10, top_n=2, no reranker, mistral-small, 512 tokens)													
3-1	Recursive	320	48	rag_tuned_mnr_1ep	similarity	10	None	2	mistral-small	512	baseline prompt	0.572	0.752
3-2	Recursive	320	48	rag_tuned_mnr_5ep	similarity	10	None	2	mistral-small	512	baseline prompt	0.646	0.788
3-3	Recursive	320	48	rag_tuned_mnr_10ep	similarity	10	None	2	mistral-small	512	baseline prompt	0.639	0.691
3-4	Recursive	320	48	rag_tuned_mnr_50ep	similarity	10	None	2	mistral-small	512	baseline prompt	0.664	0.764
3-5	Recursive	320	48	rag_tuned_mnr_100ep	similarity	10	None	2	mistral-small	512	baseline prompt	0.692	0.721
3-6	Recursive	320	48	rag_tuned_con_100ep	similarity	10	None	2	mistral-small	512	baseline prompt	0.563	0.788
Run 4 — Embedding Model & Reranker Combination Test (chunk=512, overlap=48, k=10, top_n=2, mistral-small, 512 tokens)													
4-1	Recursive	512	48	bge-small	similarity	10	CrossEncoder (L6-v2)	2	mistral-small	512	baseline prompt	0.648	0.819
4-2	Recursive	512	48	bge-small	similarity	10	CrossEncoder (L-12-v2)	2	mistral-small	512	baseline prompt	0.680	0.822
4-3	Recursive	512	48	bge-small	similarity	10	CrossEncoder (electra-base)	2	mistral-small	512	baseline prompt	0.624	0.806
4-4	Recursive	512	48	bge-base	similarity	10	CrossEncoder (L6-v2)	2	mistral-small	512	baseline prompt	0.613	0.859
4-5	Recursive	512	48	bge-base	similarity	10	CrossEncoder (L-12-v2)	2	mistral-small	512	baseline prompt	0.650	**0.861**
4-6	Recursive	512	48	bge-base	similarity	10	CrossEncoder (electra-base)	2	mistral-small	512	baseline prompt	0.599	0.828
4-7	Recursive	512	48	bge-large	similarity	10	CrossEncoder (L6-v2)	2	mistral-small	512	baseline prompt	0.654	0.806
4-8	Recursive	512	48	bge-large	similarity	10	CrossEncoder (L-12-v2)	2	mistral-small	512	baseline prompt	**0.703**	0.819
4-9	Recursive	512	48	bge-large	similarity	10	CrossEncoder (electra-base)	2	mistral-small	512	baseline prompt	0.630	0.867
Run 5 — Chunk Size & top_n Grid (bge-large, k=10, CrossEncoder L-12-v2, claude-sonnet-4-6, 512 tokens)													
5-1	Recursive	256	48	bge-large	similarity	10	CrossEncoder (L-12-v2)	1	claude-sonnet-4-6	512	baseline prompt	0.816	0.593
5-2	Recursive	256	48	bge-large	similarity	10	CrossEncoder (L-12-v2)	2	claude-sonnet-4-6	512	baseline prompt	0.720	0.833
5-3	Recursive	384	48	bge-large	similarity	10	CrossEncoder (L-12-v2)	1	claude-sonnet-4-6	512	baseline prompt	0.810	0.567
5-4	Recursive	384	48	bge-large	similarity	10	CrossEncoder (L-12-v2)	2	claude-sonnet-4-6	512	baseline prompt	0.699	0.849
5-5	Recursive	512	48	bge-large	similarity	10	CrossEncoder (L-12-v2)	1	claude-sonnet-4-6	512	baseline prompt	**0.749**	0.834
5-6	Recursive	512	48	bge-large	similarity	10	CrossEncoder (L-12-v2)	2	claude-sonnet-4-6	512	baseline prompt	0.702	**0.859**
Run 6 — Search Type Test (bge-large, chunk=512, overlap=48, k=10, CrossEncoder L-12-v2, top_n=2, mistral-small, 512 tokens)													
6-1	Recursive	512	48	bge-large	similarity	10	CrossEncoder (L-12-v2)	2	mistral-small	512	baseline prompt	0.703	0.822
6-2	Recursive	512	48	bge-large	MMR (fetch_k=20)	10	CrossEncoder (L-12-v2)	2	mistral-small	512	baseline prompt	0.659	0.782
Run 7 — Splitter Comparison (bge-large, chunk=512, k=10, CrossEncoder L-12-v2, mistral-small, 512 tokens)													
7-1	Recursive	512	48	bge-large	similarity	10	CrossEncoder (L-12-v2)	1	mistral-small	512	baseline prompt	0.749	0.784
7-2	Recursive	512	48	bge-large	similarity	10	CrossEncoder (L-12-v2)	2	mistral-small	512	baseline prompt	0.703	0.812
7-3	RST	512	32*	bge-large	similarity	10	CrossEncoder (L-12-v2)	1	mistral-small	512	baseline prompt	**0.794**	0.797
7-4	RST	512	32*	bge-large	similarity	10	CrossEncoder (L-12-v2)	2	mistral-small	512	baseline prompt	0.640	0.821
Run 8 — Generator Model & Max Tokens Test (RST splitter, bge-large, chunk=512, k=10, CrossEncoder L-12-v2, top_n=1)													
8-1	RST	512	32*	bge-large	similarity	10	CrossEncoder (L-12-v2)	1	mistral-small	512	baseline prompt	0.794	0.785
8-2	RST	512	32*	bge-large	similarity	10	CrossEncoder (L-12-v2)	1	claude-sonnet-4-6	256	baseline prompt	0.794	0.796
8-3	RST	512	32*	bge-large	similarity	10	CrossEncoder (L-12-v2)	1	claude-sonnet-4-6	384	baseline prompt	0.794	0.844
8-4	RST	512	32*	bge-large	similarity	10	CrossEncoder (L-12-v2)	1	claude-sonnet-4-6	512	baseline prompt	0.794	**0.856**
8-5	RST	512	32*	bge-large	similarity	10	CrossEncoder (L-12-v2)	1	claude-sonnet-4-6	1024	baseline prompt	0.794	0.837
8-6	RST	512	32*	bge-large	similarity	10	CrossEncoder (L-12-v2)	1	api-gpt-oss-120b	512	baseline prompt	0.794	0.784
8-7	RST	512	32*	bge-large	similarity	10	CrossEncoder (L-12-v2)	1	api-gpt-oss-120b	1024	baseline prompt	0.794	0.796
Run 9 — Prompt Design Test (RST splitter, bge-large, chunk=512, k=10, CrossEncoder L-12-v2, claude-sonnet-4-6, 512 tokens)													
9-1	RST	512	32*	bge-large	similarity	10	CrossEncoder (L-12-v2)	1	claude-sonnet-4-6	512	new instruction, context-first	0.794	0.840
9-2	RST	512	32*	bge-large	similarity	10	CrossEncoder (L-12-v2)	1	claude-sonnet-4-6	512	new instruction, question-first	**0.794**	**0.876**

Run 1 - Baseline:

We started with a baseline to make sure the pipeline worked. We used default models and with no reranker to establish a reference point before any optimization.

Run 2 - Reranker Test:

Motivated by the low precision, we added a CrossEncoderReranker (ms-macro-MiniLM-L6-v2) with top_n=3 and used RFGridSearch over k = {5, 10} to measure sensitivity to candidate pool size.

Run 3 - Fine-tuned Embedding Model Test:

In order to further improve the retrieval score, we tested the performance of our different tuned embedding models (experimenting with Multiple Negatives Ranking Loss and Contrastive Loss) over different epochs (1, 5, 10, 50, 100) and over different top-k values (2,3). For training the embedding models, we combined our golden Q&A pairs and the golden validation Q&A pairs and then split it 50-50 to get train and test sets. We turned off the re-ranker for these experiments as it seemed to hurt performance here. We are showing the results with chunk_size = 320 and chunk_overlap=48 as this setting gave promising results.

We see that the MNRL tuned models seem to perform best (at 100 epochs). Generation and recall was better for top_k=3, while precision was better at k=2.

Run 4 - Embedding Model & Reranker Combination Test:

We expanded the embedding search to off-the-shelf models and tested over 3 embedding sizes (bge-small, bge-base, bge-large) x 3 reranker models (ms-marco-MiniLM-L6-v2, ms-marco-MiniLM-L-12-v2, ms-marco-electra-base), producing 9 configs.
The combination of bge-large and ms-marco-MiniLM-L-12-v2 achieved the best retrieval score (0.704) and good enough generation score (0.819), and was selected as the embedding and reranker for all subsequent runs.

Run 5 - Chunk Size & top_n Combination Test:

With the embedding and reranker fixed, we tested over chunk_size = {256, 384, 512} x top_n = {1, 2}. Sizes chosen to span from high-specificity (256) to section-level (512). top_n = {1, 2} directly measures the retrieval-generation tradeoff.

Run 6 - Search Type Test:

We compared similarity search vs. MMR and similarity search consistently outperformed MMR. So we fixed the search type to similarity.

Run 7 - Splitter Test:

We observed that `RecursiveCharacterTextSplitter` uses fixed token counts to split documents, which can cut mid-section and mix content from different topics into the same chunk. Since the RapidFire AI documentation is structured in RST format with well-defined boundaries, we designed a custom `RSTSectionSplitter` that detects these boundaries using regex and splits at the section level, preserving each section's title within the chunk.
For sections exceeding 512 tokens, it falls back to `RecursiveCharacterTextSplitter` with chunk_size=512 and overlap=32 to stay within the embedding model's effective range.
*(Our plan was to set overlap to 48, but there was a mistake in the code which we found out later. So the experiment we ran at that time was actually setting overlap to 32.)
We tested over (Recursive vs. RST) x top_n={1,2}. The RST splitter with top_n=1 achieved a new highest retrieval score of 0.794, as each chunk now corresponds to a semantically complete section, improving the precision from 0.778 to 0.822.

Run 8 - Generator Model & Max Tokens Test:

With the retrieval part finalized, we tested over different generator models (mistral-small, claude-sonnect-4-6, api-gpt-oss-120b) and max_completion_tokens = {256, 384, 512, 1024}, , spanning different model capability tiers and output length budgets.

Run 9 - Prompt Design Test:

We tested a more constrained system instruction vs. the baseline prompt, and compared context-first vs. question-first user message format. Question-first format is motivated by the hypothesis that seeing the question first helps the model retrieve relevant information more selectively.

Bug Fix and Additional Test:

After completing the main experiments, we identified two issues in the RSTSectionSplitter implementation: (1) the fallback threshold used word count instead of tiktoken token count, causing some long sections to bypass fallback splitting; (2) the actual fallback overlap was 32 tokens due to a code bug, not 48 as originally intended.

After fixing both issues, we ran additional configs with the corrected implementation over overlap $\in \{48, 64\}$ with claude-sonnet-4-6 as the judge. The best result was overlap=64, chunk_size=512, top_n=1, achieving retrieval score=0.832 and generation score=0.874. The leaderboard score improved from 0.835 to 0.853. We adopted this as the final pipeline configuration.

4. Results and Analysis

How We Arrived at the Final Pipeline:

We optimized the pipeline sequentially: first fixing retrieval knobs (reranker → embedding model → chunk size → splitter), then optimizing generation knobs (generator model → prompt format). Each step used the best config from the previous step as the new baseline, culminating in the final configuration after the bug fix.

Retrieval knobs:

The reranker and splitter had the largest impact. Adding CrossEncoder reranking (Run 2) raised Retrieval Score from 0.443 to 0.526, driven by precision rising from 0.213 to 0.348. Switching to RSTSectionSplitter (Run 7) further pushed retrieval score from 0.749 to 0.794, as semantically complete chunks improved precision from 0.778 to 0.822. Upgrading the embedding model to bge-large (Runs 3-4) also contributed substantially.

Regarding chunk_size and top_n, results (Run 5) showed that top_n=1 gives higher retrieval score due to improved precision, while top_n=2 gives higher generation score due to more context available to the generator. Among all the configs, top_n=1 and chunk_size=512 achieved the best balance, and was selected as the chunk size for subsequent runs.

Generation knobs:

Generator model choice had the largest single impact: switching from mistral-small to claude-sonnet-4-6 improved Generation Score from 0.785 to 0.856. Increasing maximum completion tokens beyond 512 didn't help and slightly hurt performance. The question-first format improved the generation score from 0.840 to 0.876 by helping the model focus on what to look for before reading the context.

We also experimented with adding few-shot examples to the prompt, but this unexpectedly hurt generation score.

Retrieval vs. Generation tradeoff:

A consistent tradeoff appeared across experiments: top_n=1 always yielded higher retrieval score (better precision) while top_n=2 yielded higher generation score (more context). Smaller chunk sizes similarly improved retrieval score but hurt generation score when combined with top_n=1. We selected top_n=1 with chunk_size=512 as it maximizes the combined leaderboard score.

Future work:
With more time and budget, we would test hybrid retrieval methods to see if it could further improve retrieval score. We also want to spend more time on prompt engineering to improve the generation score.

5. AI Tools Statement

AI was used to help with the generation of the golden dataset, as well as for coding. We used Cursor and Claude as coding assistants to iteratively help with writing the code and fixing bugs for our golden Q&A generator, as well as the experimental pipeline and main.py file.