Data Processing

For data cleaning and processing, we first removed all keys in the JSON files which contained “css”, “privacy” or “disclaimer” as they consisted of unwanted information. We then removed non-ASCII characters as well as duplicate sections throughout the file to reduce the file size as much as possible.

Knowledge Base Design

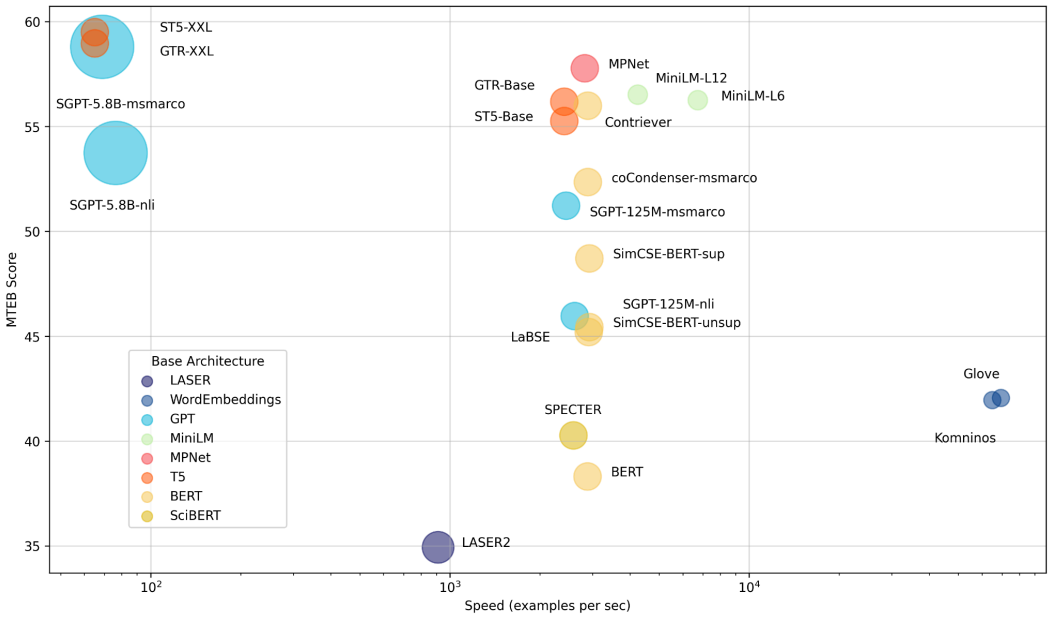
After evaluating various database technologies (vector databases, graph databases, relational databases, and NoSQL) we chose a vector database to power our RAG agent. The core reason was the system’s primary goal: retrieving information based on the semantic meaning of user queries, not exact keyword matches. As relational databases are not known for their semantic abilities, and due to the fact that our data was highly unstructured, we decided against relational databases. Because graph databases require explicit modeling of entities and relationships, and our data was not fit for that, they did not align with our content. Storing and querying text data in a NoSQL database would require extra work, like implementing a custom search layer, and even then, NoSQL databases wouldn’t natively offer the same efficiency in retrieving semantically relevant information as a vector database.

In contrast, vector databases such as Chroma are optimized for semantic search. Their ability to store and efficiently query high-dimensional embeddings made them the ideal fit for our use case, and we ended up deciding to use Chroma DB.

For chunking, we initially tried the simplest way of chunking the large textual data using character text splitting, however, we realized that this was not optimal as this technique splits text halfway into the word and therefore loses context and meaningful information for the model. After carefully reviewing several approaches for chunking for RAG systems, our decision was influenced primarily by the benchmarking results provided in the Chroma Technical Report on Evaluating Chunking Strategies for Retrieval. In this report, we initially wanted to do Cluster Semantic Chunking as it achieved the highest metrics for all-MiniLM-L6-v2 model (which is what we used as our embedding model), with a score in Precision (7.2 ± 6.1), PrecisionΩ (33.6 ± 20.0), and IoU (7.2 ± 6.0) shown on the table below. Unfortunately, we ran out of computing power to finish chunking with cluster semantic approach, therefore, we proceeded to use recursive character text split instead.

Chunking	Size	Overlap	Recall	Precision	Precision$_\Omega$	IoU
Recursive	250 (~180)	125	78.7 ± 39.2	4.9 ± 4.5	21.9 ± 14.9	4.9 ± 4.4
TokenText	250	125	**82.4 ± 36.2**	3.6 ± 3.1	11.4 ± 6.6	3.5 ± 3.1
Recursive	250 (~162)	0	78.5 ± 39.5	5.4 ± 4.9	26.7 ± 18.3	5.4 ± 4.9
TokenText	250	0	77.1 ± 39.3	3.3 ± 3.0	16.4 ± 10.3	3.3 ± 3.0
Recursive	200 (~128)	0	75.7 ± 40.7	6.5 ± 6.2	31.2 ± 18.4	6.5 ± 6.1
TokenText	200	0	76.6 ± 38.8	4.1 ± 3.7	19.1 ± 11.0	4.1 ± 3.6
■● Cluster	250 (~168)	0	77.3 ± 38.6	6.1 ± 5.1	28.6 ± 16.7	6.0 ± 5.1
■● Cluster	200 (~102)	0	75.2 ± 39.9	**7.2 ± 6.1**	**33.6 ± 20.0**	**7.2 ± 6.0**

Moreover, we used the [Massive Text Embedding Benchmark (MTEB)] as the main basis for choosing our embedding model. The diagram below from METB illustrates that all-MiniLM-L6-v2 provides a good balance between speed and performance among all embedding models. While larger models like GTR-XXL or SGPT models offer maximum performance, they also tend to produce bigger embeddings which is not practical for our case due to a limited storage capacity.



Retrieval System

In order to try and optimize the retrieval process, we tried implementing a filter system based on metadata. The idea was to assign labels to text chunks using named entity recognition (NER), and then extract similar labels from the user query. This way, we could restrict retrieval to only those embeddings whose metadata matched the query's labels, improving efficiency. Initially, we experimented with high-performing NER models such as DistilBERT-NER and Flair. However,

due to the large number and size of our chunks, combined with the complexity of these models, both models proved too slow in practice. When we attempted the same procedure with spaCy, the process was much faster. Despite that, when tested on a small number of embeddings, spaCy's performance was undesirably low, so we ultimately chose not to include this approach in our final implementation, as it appeared to hinder overall performance.

We also considered using our LLM to rewrite the user's query in a way that would better align with the structure of our RAG system. The motivation behind this was that user-generated queries may not always match the form or phrasing of the embedded chunks where the relevant information resides. Unfortunately, we did not have the time to implement this idea.

Additionally, we explored the idea of ranking the retrieved results before passing them to the LLM, with the goal of improving both relevance and efficiency. However, due to time constraints, this approach also remained unimplemented.