Bookmark Assistant Bot:
AI-Powered Bookmark Management System

Xiaoke Liu
UIUC
Champaign, IL, USA
xiaokel4@illinois.edu

Qi Jing
UIUC
Champaign, IL, USA
qijing3@illinois.edu

Congzheng Chen
UIUC
Champaign, IL, USA
cc186@illinois.edu

Xiu Dong
UIUC
Champaign, IL, USA
xiudong2@illinois.edu

1 BACKGROUND

1.1 Motivation

Many users frequently save social media posts as a way to "come back later", but this rarely happens due to poor organization, lack of memory, or volume overload. Our bot addresses this common “digital forgetfulness” by turning saved content into searchable knowledge. It serves a real need by offering a lightweight personal memory system, increasing productivity, reducing time spent re-finding content, and promoting better knowledge organization.

1.2 Intended Users

The target users are frequent social media users who habitually save content but often forget or fail to revisit it.

1.3 Major Functions

We propose to develop a “Bookmark Assistant Bot”, a tool that helps users manage and retrieve their saved posts on social platforms like RedNote, Instagram, or similar apps. The envisioned tool will be a standalone web-based application that allows users to (1) forward useful posts to the bot, (2) automatically extract and index key information using NLP, and (3) retrieve saved content using natural language queries later. The main functions will include content ingestion (via forwarding links, screenshots, or text), intelligent tagging and summarization, and semantic search.

2 IMPLEMENTATION

2.1 System Architecture

Our system follows a modular architecture centered around a Retrieval-Augmented Generation (RAG) framework, as shown in the diagram below. It is composed of four key layers: Frontend, RAG, Storage and Data.

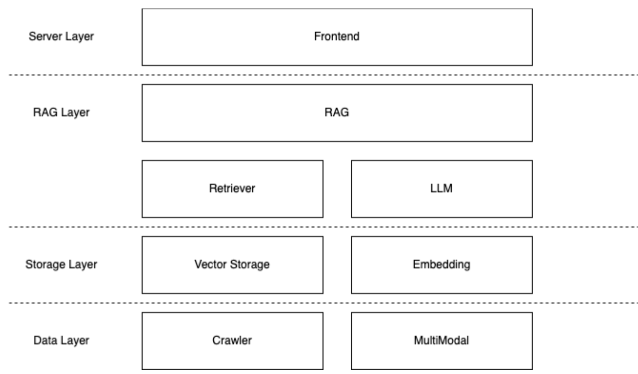


Figure 1: Overall System Architecture

At the top layer, we use a Gradio-based frontend, which provides an intuitive interface for users to enter queries, upload Rednote links. This allows seamless interaction with the underlying pipeline, even for non-technical users.

The core of the system is the RAG module, which is built based on semantic retrieval and language generation. Given a user query, it performs dense retrieval of relevant RedNote posts and feeds the contextual results to a large language model (LLM) to generate fluent and grounded responses.

The backend is composed of the following modular components:

- **Retriever**: Powered by multi-query expansion using Gemini, it formulates semantically diverse sub-queries to improve recall and robustness.
- **LLM**: Google’s Gemini API is used for both multi-query generation and response synthesis, ensuring contextual relevance and fluency.
- **Vector Storage**: Embeddings of RedNote content are stored in Milvus Lite, a lightweight and efficient vector database that enables real-time semantic search.
- **Embedding**: We use SentenceTransformers (SBERT) to convert posts and queries into 384-dimensional embeddings that capture deep semantic meaning.

- **Crawler**: A custom scraping module is used to collect RedNote content, including titles, text, images, and url.
- **Multimodal Support**: The system also incorporate OCR (e.g., Tesseract) in the final version, allowing users to extract text from images in the post to improve search scope and precision.

2.2 Data Layer

2.1.1 **Crawler.**

We build a robust web crawler for scraping Rednote board posts, implemented primarily with Playwright. We launch a persistent Chromium context so cookies and login tokens survive across runs, then navigate to the homepage and check for a "log in" button. If it appears, we pause for manual authentication before proceeding—this hybrid approach lets us bypass anti-bot measures without interrupting the crawl.

Once our session is valid, we load the board URL and trigger its infinite-scroll loader by repeatedly scrolling to the page bottom. On each iteration, we run a single page.evaluate() call to grab all <a> hrefs under the note-item selector, prepend the domain, dedupe via a JavaScript Set, and accumulate unique URLs. Batching link extraction in this way minimizes Python↔browser context switches and avoids race conditions as new posts load.

With the full URL list collected, we visit each note using page.goto(..., wait_until='domcontentloaded') and wait explicitly for .title and .note-text. Inside the page, we extract every <img> src, filter out avatars, strip query parameters, and dedupe by base path. Wrapped in a retry loop with brief pauses, this ensures resilience against timeouts or layout shifts. Finally, we assemble results into a Pandas DataFrame, drop duplicates by URL, and export to Excel—using tqdm bars to give real-time feedback on both scrolling and detail-scrape phases.

2.1.2 **Multimodal**

Building on the previously crawled RedNote content—which includes both textual descriptions and post images—we introduce an OCR-enhanced multimodal pipeline to incorporate image-based information into downstream retrieval and generation.

For each post, we filter out non-content images (e.g., avatars, UI icons) and apply Tesseract OCR with `chi_sim+eng` language settings to extract textual information from relevant images.

The extracted text from images is treated as an additional context block and embedded using the same sentence embedding model as the textual content, ensuring alignment in vector space. During retrieval, both original and OCR-derived chunks are jointly searched, enabling the system to capture information that would otherwise be inaccessible—such as ingredient lists, brand names, or visual labels embedded in images.

This multimodal integration significantly enhances the system's ability to answer visually grounded questions and improves recall in image-heavy use cases.

2.3 Storage Layer

We adds a Milvus Lite–based storage layer for semantic indexing and retrieval. We use pymilvus to manage the database and SentenceTransformer to generate 384-dimensional text embeddings, bundling collection setup, embedding computation, data insertion, vector search, and attribute filtering into a single MilvusStorage class. Downstream applications can simply say "find me posts like this" or "delete all entries matching …."

2.4 RAG Layer

We implement a retrieval component MilvusMultiQueryRetriever that enhances query recall by generating multiple semantically varied sub-queries for a single user question. Specifically, when receiving a user's query, we first use Gemini based on ChatVertexAI to generate paraphrased or diversified versions of the original user query. This allows the retriever to account for semantic variation and improve recall. Then, all queries are converted into dense embeddings using the same model used for indexing. For each generated query, the system retrieves top-k documents from a Milvus vector database by cosine similarity. Finally, all candidate results are merged, duplicates are removed (based on document ID or content), and the final results are returned.

We then integrate all components—retriever, vector store, and LLM—into a single retrieval-augmented generation (RAG) pipeline using LangChain. For retriever, we utilize the above MilvusMultiQueryRetriever to find relevant RedNote content from the vector store. For vector storage, we rely on the MilvusStorage for embedding storage and similarity search. For LLM integration, we use ChatVertexAI to generate final responses based on the retrieved context. And we also carefully design prompt instructs to the LLM.

A typical workflow is:

1 User submits a question.
2 The retriever expands the query and performs multi-vector search.
3 Top results are formatted and passed as input context.
4 The LLM generates a grounded, context-aware response.

2.5 Server Layer

We use Gradio as the frontend framework to build an interactive chat interface for a Retrieval-Augmented Generation based question answering system. Gradio allows for quick development with features like responsive UI, text input, Markdown rendering, and example query support. The layout is built using gr.Blocks and gr.Row, forming a clean two-column structure with components such as a textbox for inputting Rednote collection links and a chat interface for real-time conversation. The frontend is integrated with a backend powered by Vertex AI and Milvus, enabling full RAG pipeline functionality. This tech stack offers advantages such as low-code implementation, fast local and cloud deployment, high extensibility, and a user-friendly experience. The system only requires a Python environment, Gradio dependencies, and internet access to connect to external services.

3 ABLATION AND CONTRAST

3.1 Prompt

We evaluated the impact of prompt design on answer quality, particularly in terms of completeness, factual grounding, and style. Three versions of the prompt were tested:

Strict Context-Only Prompt: Instructed the model to rely only on retrieved context and explicitly respond with “I don’t know” if information was missing.

Lenient Completion Prompt: Allowed the model to supplement incomplete context with general knowledge if needed.

Final Version (Context + Prioritized Supplement): Prioritized context first, permitted reasonable general knowledge supplementation, and explicitly emphasized brevity, clarity, and direct extraction from bookmarks.

We tested these prompt variations on a steamed dumpling recipe query, where the post’s text lacked useful instructions and most of the content was image-based (Figure 2).

The strict prompt produced safe but overly terse answers, omitting valuable inferred steps. The lenient version included helpful elaboration but occasionally hallucinated details (e.g., incorrect ingredient names). In contrast, the final prompt struck the best balance: it generated grounded responses that combined OCR-extracted instructions with high-precision formatting and minimal hallucination.

This shows that carefully engineered prompt structure, especially those that (1) prioritize grounded context, (2) allow fallback reasoning only when necessary, and (3) explicitly enforce response style guidelines, can significantly improve the relevance and reliability of LLM answers in RAG systems.

3.2 Multimodal

To evaluate the contribution of multimodal integration, we conducted an ablation study on RedNote posts where key information is presented primarily in images. One representative example involves a post about steamed dumplings, in which the main text contained minimal instructional content—critical details such as ingredients, quantities, and cooking steps were embedded solely within an image.

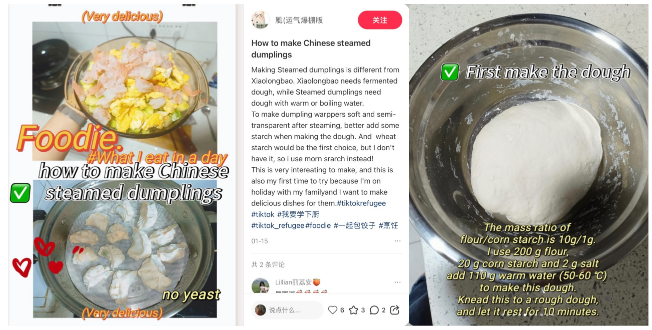


Figure 2: Post Example

Prior to incorporating OCR-based image text extraction, the generated answer would typically include high-level steps such as “prepare the dough and steam it,” lacking actionable details.

After enabling image text extraction via Tesseract OCR and including the results in our embedding and retrieval pipeline, the system showed a marked improvement. The generated responses now included detailed information such as precise ingredient quantities, step-by-step preparation instructions, and even specific timing suggestions (e.g., “rest the dough for 20 minutes”).

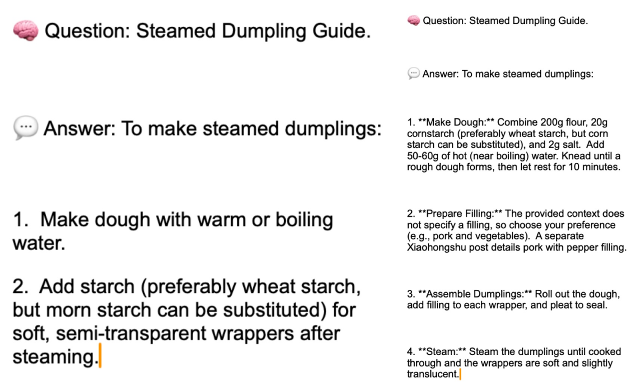


Figure 3: Comparison before and after multimodal addition (left: before addition; right: after joining)

This contrast demonstrates the practical effectiveness of our multimodal pipeline. By leveraging OCR to recover hidden image-based content, we are able to significantly enrich the knowledge base and improve the informativeness and grounding of generated answers in real-world scenarios.

4 EVALUATION RESULTS

To comprehensively evaluate our system, we conducted a multidimensional analysis across three key aspects: retrieval efficiency, generation quality, and response time.

- **Retrieval Efficiency** assesses how effectively the system surfaces relevant documents given a user query. We measured this using metrics such as precision and recall, allowing us to quantify both the relevance and completeness of the retriever's output across diverse query types.

- **Generation Quality** evaluates the final output produced by the language model, focusing on fluency, factual accuracy, contextual grounding, and informativeness. This was assessed through both manual inspection and alignment with the retrieved context.
- **Response Time** captures the end-to-end latency of the RAG pipeline, from query input to final answer. This dimension helps determine whether the system meets real-time or near-real-time usability requirements, which is particularly important for interactive applications.

Together, these three dimensions provide a holistic view of system performance and help identify areas for future optimization.

4.1 Retrieval Efficiency

To systematically evaluate the retrieval performance of our RAG pipeline, we designed a small-scale, qualitative evaluation based on four representative query types. Each query is manually crafted to reflect a distinct retrieval challenge, allowing us to assess the system's behavior across a range of realistic scenarios.

For each query, we manually annotate a set of relevant document IDs. The retriever then returns the top-k results, and we compute standard information retrieval metrics of precision and recall.

Query Type	Query Text	Precision before LLM	Recall before LLM	Precision after LLM	Recall after LLM
Relevant Documents Exist	tea recipe	0.6	0.75	1	0.75
No Relevant Documents	fishing tips	0	1	1	1
Broad / Vague Query	recipe	0.6	0.83	1	0.83
Highly Specific Query	China Visa Guide for Foreign Nationals	0.17	0.5	1	0.5

Table 1: Precision and Recall before and after LLM for representative queries

After incorporating the LLM step, all queries achieved a precision of 1.00. This indicates that the LLM is highly effective in filtering or re-ranking the retrieved results, ensuring that only relevant documents are surfaced in the final output—even for noisy or semantically ambiguous queries.

Recall did not change for any query. This suggests that the LLM does not contribute to discovering new relevant documents but rather acts as a re-ranking or filtering layer on top of the retriever's initial candidates. This behavior is expected in most RAG systems where LLMs do not retrieve directly from the corpus.

These results highlight the complementary roles of retrievers and LLMs in RAG systems: while retrievers ensure semantic coverage, LLMs act as powerful filters to enhance final answer quality. However, the unchanged recall also suggests room for improvement in the retriever component—potentially through better embeddings, query expansion, or tuning retrieval parameters.

4.2 Generation Quality

The goal of this part is to evaluate the quality of answers generated by a RAG model using Gemini. The evaluation focuses on different types of queries, which are divided into two main categories:

1. RAG-Based Questions (queries that rely on document retrieval)

1.1. Simple Questions (factual information that can be directly extracted from the retrieved documents)

Q1 "What are the must-see places in Hangzhou?"

1.2. Reasoning Questions

Q2 "How can international shoppers find high-quality Taobao products?"

1.3. Aggregation Questions(aggregate information from multiple documents)

Q3 "What are the top 5 recommendations for learning Mandarin from different Rednote posts?"

2. Non-RAG Questions (queries that do not rely on document retrieval)

Q4 "How do you solve a Rubik's Cube?"

For each generated answer, the following criteria will be used to evaluate the quality:

- Accuracy: Does the answer correctly reflect the information requested?
- Relevance: Does the answer stay on topic and directly address the user's query?
- Completeness: Does the answer cover all required points without omissions?
- Usefulness: Is the answer actionable or informative for the user?
- Fluency: Is the answer coherent, well-structured, and easy to understand?

Query ID	Query Type	Accuracy	Relevance	Completeness	Usefulness	Fluency

Q1	RAG-Based Simple Questions	3/5	5/5	4/5	4/5	5/5
Q2	RAG-Based Reasoning Question	5/5	1/5	1/5	1/5	5/5
Q3	RAG-Based Aggregation Questions	4/5	5/5	4/5	4/5	5/5
Q4	Non-RAG Questions	5/5	1/5	1/5	1/5	5/5

Across four evaluation experiments, our RAG system demonstrated consistently excellent fluency and solid accuracy whenever relevant sources were found. It confidently answered fact-based queries with clarity and useful detail, highlighting its robust language capabilities. When a query falls outside the user's collection, it now simply states "I don't know" and informs the user that no matching content exists—an honest fallback that cuts down on hallucinations and sets clear expectations. In future iterations, we can offer a general answer in such cases if desired. Coupled with strict citation checks, trusted resource suggestions, and concrete examples, our system is evolving from a polished generator into a truly dependable, comprehensive assistant.

4.3 Response Time

To systematically evaluate the retrieval performance of our RAG pipeline, we designed a small-scale, qualitative evaluation based on four representative query types. Each query is manually crafted to reflect a distinct retrieval challenge, allowing us to assess the system's behavior across a range of realistic scenarios. To evaluate the performance of our chatbot under different input loads, we compared its response speed using two RedNote collection sizes: a small collection with 3 posts and a large one with 30 posts. We measured both the initial response time, which includes the full RAG pipeline (web scraping, vector embedding, and storage into the vector database), and the follow-up response time, which only involves querying the language model using pre-built vectors.
We tried 10 times for each of the two collection sizes, and here are the average times for each attempt:

Collection Size	Initial Response Time (with RAG processing)	Follow-up Response Time (query only)
3 posts	84.7s	7.1s
30 posts	498.1s	6.8s

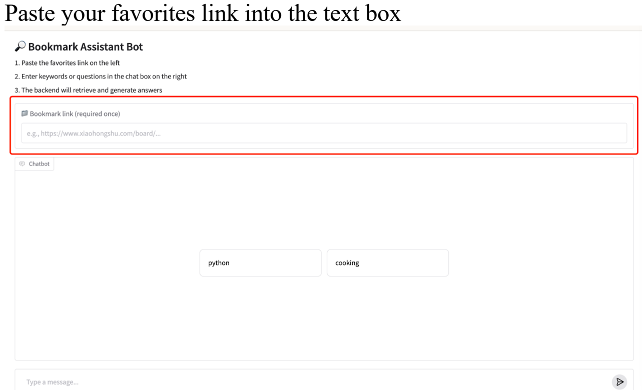
Analysis:

1. The initial response time increases significantly with the size of the collection, reaching nearly 500 seconds for 30 posts. This is expected, as more posts require more extensive web scraping, vector embedding, and database storage operations.
2. In contrast, the follow-up response time remains low and stable across both scenarios. This is because once the vector database has been constructed, subsequent questions bypass the preprocessing steps and rely solely on retrieval and generation, which are efficient.

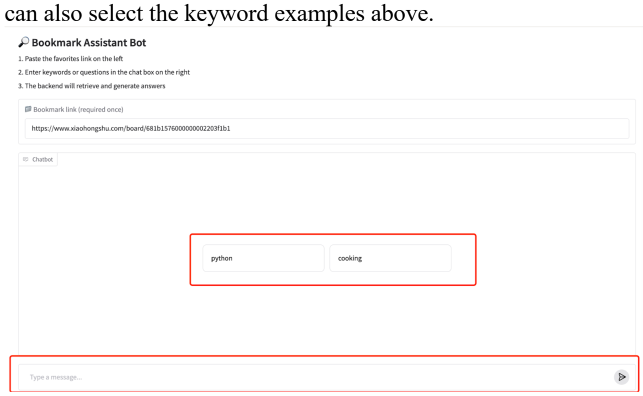
In conclusion, the system's initial latency is sensitive to collection size due to the overhead of the RAG setup process. However, once the setup is complete, the chatbot provides consistently fast and responsive follow-up interactions, regardless of the original collection size.

5 USER CASE

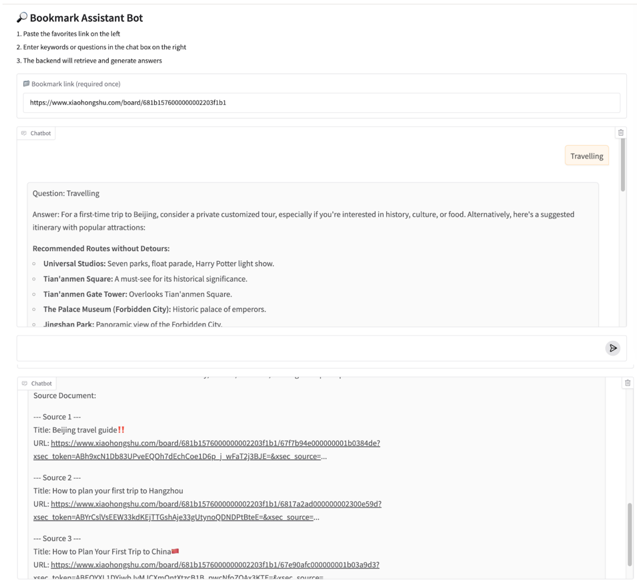
Step1:
Paste your favorites link into the text box



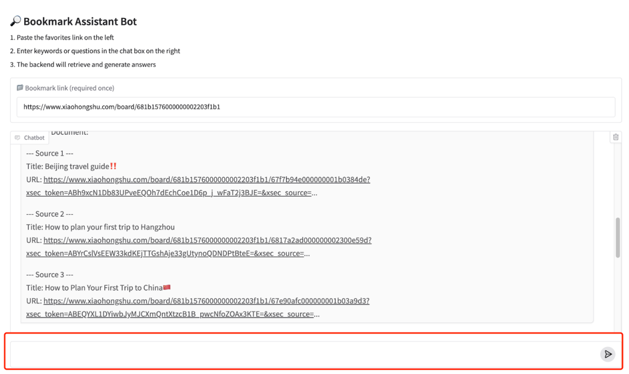
Step2:
Enter keywords or questions you want to ask in the chat box. You can also select the keyword examples above.



Then, the chatbot will give the answer to this question/keyword and list the title and url of related sources.



Step3 (optional):
After the chatbot gives an answer, you can continue to ask questions in the chat box.



Then it will give further answer to this following question/keyword and list the title and url of related sources.