Personal Research Portal Final Report

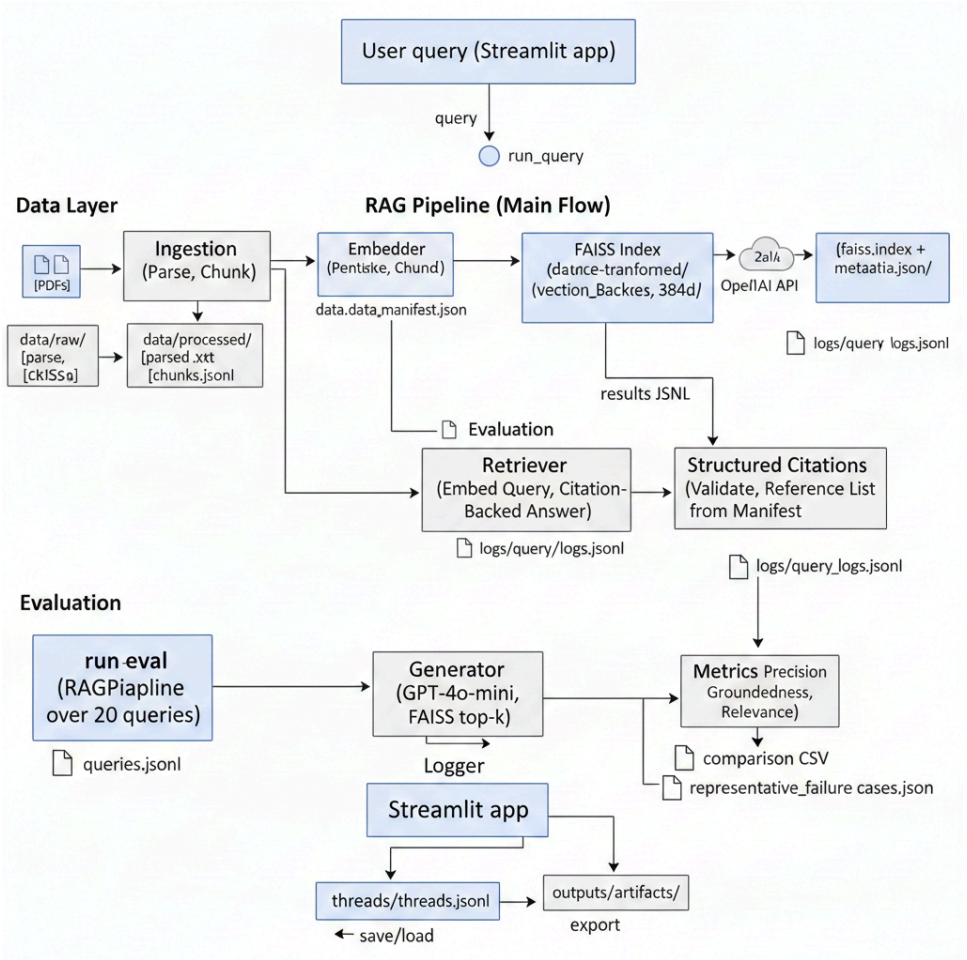
by Evelynn Chen (evelynnc)

Introduction

The Personal Research Portal is a web application for sleep and well-being research. The research domain is sleep patterns, quality, duration, and their effects on physical health, mental well-being, and cognitive performance, using a corpus of 16 peer-reviewed papers (525 chunks). Using this portal, you can go from a research question to a grounded synthesis by 1) asking questions and receiving citation-backed answers, (2) browsing saved research threads, (3) generating and exporting evidence tables (CSV, Markdown, PDF), and (4) viewing and re-running evaluation metrics from Phase 2.

Architecture

The portal is a multi-page Streamlit app with four pages (Search, History, Artifacts, and Evaluation), and utlizes a shared utilities module. All paths are relative to the repo root (the directory that contains src/, data/, outputs/, logs/, threads/). The main data, vector index, and RAG pipeline was built in Phase 2 while Phase 3 added a Streamlit UI layer for a better user experience when interacting with the RAG pipeline.



The diagram explains mainly the backend (Phase 2) part of the portal.

Data layer:

- data/raw/ contains the 16 raw source PDFs which are ingested by src/ingest/pipeline.py to parse the PDFs and save them as data/processed/{source_id}.txt
- data/processed/chunks.jsonl has one JSON object per line containing chunk_id, source_id, text, token_count, start_char, end_char.
- data/data_manifest.json has the JSON list of source metadata keyed by id (source_id): title, authors, year, source_type, link_or_DOI, relevance_note, raw_path, and processed_path. This is used for citation resolution and reference list generation.

Vector layer:

- Embedder (src/rag/embedder.py): Sentence-transformers all-MiniLM-L6-v2; embeds chunk text into 384-dimensional vectors, normalized for cosine similarity.
- Vector store (src/rag/vector_store.py): FAISS IndexFlatIP (inner product = cosine when normalized). Stores embeddings and chunk metadata; supports search(query_embedding, k) returning top-k chunks with similarity_score.
- Index on disk (data/processed/vector_index/): faiss.index and metadata.json (list of chunk dicts). Built by src/rag/build_index.py: load chunks.jsonl → embed → add to FAISS → save.

RAG pipeline:

- Retriever (src/rag/retriever.py): Loads the FAISS index and metadata from data/processed/vector_index/. Embeds the user query with the same model, runs FAISS search (top-k=5), returns list of chunk dicts with chunk_id, source_id, text, similarity_score.
- Generator (src/rag/generator.py): utilizies GPT-4o-mini with a system prompt that enforces citation format (source_id, chunk_id) and instructs the model to state when evidence is insufficient. Takes query + retrieved chunks, returns answer, model/prompt metadata, token usage.
- Structured citations (src/rag/structured_citations.py) [enhancement] Reads data/data_manifest.json, extracts citations from the answer, validates each (source_id, chunk_id) against retrieved chunk IDs, and builds a formatted reference list from the manifest. Adds reference_list, citation_validation_passed, invalid_citations, num_unique_sources to the result.
- Logger (src/rag/logger.py): Appends each query result (query, answer, retrieved_chunks, metadata) as one JSON line to logs/query_logs.jsonl.
- Pipeline entry (src/rag/pipeline.py): RAGPipeline composes Retriever → Generator → StructuredCitationGenerator → Logger. query(query, k=5, log=True, enhance=True) returns the full result dict.

Evaluation:

- src/eval/queries.json: 20 queries (direct, synthesis, edge-case) with id, query, category, optional phase1_task.
- src/eval/run_eval.py: Loads queries, instantiates RAGPipeline, runs each query (baseline or enhanced), appends each result to outputs/baseline_eval_results.jsonl or outputs/enhanced_eval_results.jsonl. All runs also log to logs/query_logs.jsonl.

- src/eval/metrics.py: Reads a results JSONL, computes citation precision (valid citations / total citations), and LLM-scores groundedness and answer relevance (1–4). Writes outputs/baseline_eval_scores.csv or outputs/enhanced_eval_scores.csv.
- src/eval/compare_baseline_enhanced.py: Compares baseline vs enhanced scores; writes outputs/eval_report_data.json (overall metrics, improvements, metrics_by_category, representative_failure_cases), outputs/baseline_enhanced_comparison.csv, and outputs/representative_failure_cases.json.

UI Layer:

The Streamlit app is a thin layer on top of the system described above. There's a single input from the UI: the user query (Search page). That query is passed to utils.run_query(query), which instantiates RAGPipeline and calls pipeline.query(query, k=5, log=True, enhance=True). No other UI action triggers the core RAG pipeline, as History and Artifacts consume already-saved thread data or session state.

Components:

- src/app/utils.py: load_manifest() reads data/data_manifest.json; run_query(query) calls Phase 2 RAGPipeline.query(...) and returns answer, chunks, reference_list, etc.; save_thread(...) appends to threads/threads.jsonl; load_threads() reads threads for History.
- src/app/artifact_generator.py: Builds evidence table rows from RAG output (chunks, answer, manifest); exports to CSV/Markdown/PDF and writes to outputs/artifacts/.
- Pages:
    - Search: query → run_query → display + save_thread
    - History: load_threads, generate artifact from thread
    - Artifacts: display rows, download, write to outputs/artifacts/
    - Evaluation: read outputs/*, display metrics/failures, Re-run invokes run_eval → metrics → compare from repo root

Design Choices

I chose Streamlit because it comes with a lot out of the box and its default UI is sufficient for the research portal MVP without needing any custom CSS. Streamlit's st.session_state and st.switch_page support the flow from Search/History to Artifacts with a pending artifact.

I chose structured citations as my enhancement because the baseline RAG system would generate citations but had no verification if these citations actually existed in the retrieved chunks and in Phase 1 there was a recurring issue of citations being fabricated and just plain wrong, therefore this was a prevalent issue.

I chose the all-MiniLM-L6-v2 sentence transformer model because of its speed, affordability (free), and good match for my corpus size.

I chose GPT-4o-mini as my LLM because of its balance between capability and affordability.

Evaluation

In Phase 2 we evaluated the baseline RAG to an enhanced RAG system with structured citations over 20 queries (direct, synthesis, and edge-case). The metrics we collected were citation precision, groundedness, answer relevance, and average citation count. The Evaluation page displays outputs/baseline_enhanced_comparison.csv, which includes metric, baseline value, enhanced value, difference, and percent change for citation_precision, groundedness_score, and answer_relevance. I've also put them in table form below.

Baseline vs Enhanced Comparison

Metric	Baseline	Enhanced	% Change
Citation precision	77.6%	78.5%	+1.1%
Groundedness	3.4/4	3.58/4	+5.1%
Answer relevance	3.58/4	3.73/4	+4.2%
Average citation count	4.3	3.95	-8.1%

The largest improvement can be seen in groundedness with an increase in 5.1%. The fact that citations will be validated encourages the LLM to be more careful about grounding claims in actual evidence. Citation precision improves slightly by 1.1% as the model's answers align better with the query when citations are validated. Answer relevance increased by 4.2% because the reference list allows users to see which papers support each claim without having to search through the data manifest on their own. The average citation count went down by 8.1% because the enhanced system generates fewer, higher-quality citations and discourages overciting.

Trust behaviors:
Citations on every answer are heavily enforced, with the Search page always shows a "References / citations" section either containing the reference_list from the RAG result (when enhanced=True), the list of cited chunks, or a clear "No sources cited" only when the pipeline returned no reference list and no citations. If the answer text contains the phrase "The provided evidence does not contain", the Search page shows a yellow warning box: "The corpus does not contain sufficient evidence for this claim. Suggested next step: [suggestion]." The suggestion is parsed from the model's answer when it includes a sentence about "additional sources"; otherwise a generic line is used. This makes it clear to the user that the portal is missing evidence for their inputted question and what they can do next to solve the issue. This is only triggered by "does not contain" phrasing to avoid flagging answers that do use evidence but add a mild disclaimer to add additional sources. We also utilize the enhanced RAG system by default, which means the structured citations are always utilized and that invalid citations are reported immediately.

Limitations

The first limitation is the corpus size. It only has 16 papers and 525 chunks, and queries that are not really related to the main research question and sub-questions I defined in Phase 1 may retrieve only semi-relevant chunks or just get the generic insufficient evidence response. There are also a handful of edge cases and failure cases that have a drop in citation precision and groundedness (can be found in outputs/representative_failure_cases.json). Additionally, using an LLM in the evaluation pipeline to score the metrics like groundedness and answer relevance is not as good and accurate as a real human, and the scores may change from run to run. Citation precision is more deterministic of a metric. Lastly, lots of code architecture is only built for a local context and depends heavily on following the correct folder achitecture. For example, the outputs/ directory must be at the root of the project and the research threads are saved as a JSONL in a local folder, which will not work if this project were to be commercialized and poses some data privacy issues.

Next Steps

Here are some of the next steps I could take to continue refining this research portal:

1. Expand the corpus: Manually collecting more peer-reviewed articles or building an automated web crawler/agent to scrape for sources will help strengthen the RAG's knowledge base and allow it to handle more types of queries.
2. Collect data on how users use it to drive new additional enhancements: If I were to keep track of how real users are using the portal and what types of queries they make, it would help drive what new features would be the most beneficial. For example, if people tend to put in longer more complex questions, then adding query rewriting or decomposition enhancement would be the most beneficial.
3. Improving the UI/UX: Adding UI capabilities like displaying the exact chunk that a {source, chunk_id} maps to may help with trust and a copy to clipboard alongside the export function may be beneficial.