MyVictor: A Hybrid Graph and Vector Retrieval System for UB CSE Academic Advising

Sruthisri Venkateswaran
sv94@buffalo.edu

Datta Aneesh Thota
dattaane@buffalo.edu

Saroja Vuluvabeeti
sarojavu@buffalo.edu

Deven Rahul Shah
devenrah@buffalo.edu

May 2026

Abstract

MyVictor is a domain-specific academic advising chatbot for the University at Buffalo Department of Computer Science and Engineering. The system combines two complementary retrieval methods: a `Neo4j` property graph for structured relationships and a `Qdrant` hybrid vector store for semantic document retrieval. This paper documents the implemented system, including its data acquisition pipeline, graph construction process, vector indexing strategy, runtime retrieval flow, web interface, and evaluation harness. The inspected repository contains 421 lab HTML files, 32 research-area Markdown files, 121 graduate-handbook sections, 102 faculty records, and 116 course records. The central engineering result is a practical GraphRAG-style system in which the graph path answers precise relational questions, while the vector path supplies descriptive context from departmental documents. The paper also records important implementation boundaries, including data freshness limits, HTML extraction fragility, shared in-memory web state, and the fact that a production response-safety guardrail is not yet implemented in the inspected codebase.

Keywords: retrieval-augmented generation, GraphRAG, academic advising, Neo4j, Qdrant, hybrid search, web scraping, large language models

Acknowledgments

We gratefully thank our project advisors, Dr. David Doermann and Dr. Joaquin Carbonara, for their guidance, technical feedback, and support throughout the MyVictor project. We also thank the Artificial Intelligence Innovation Laboratory (A2IL) for supporting the development environment and broader project context. We gratefully acknowledge all contributors to the broader MyVictor project effort.

1 Introduction

Academic advising questions are often simple to ask but difficult to answer from a technical perspective. A student may ask who teaches a course, what prerequisites apply, which faculty work in a research area, what a lab studies, or what a graduate policy says. The answers are distributed across many sources: faculty profiles, course catalog entries, research-area pages, lab websites, and the graduate handbook. Each source is useful, but none of them alone provides a unified advising interface.

MyVictor addresses this problem by building a retrieval-augmented generation system grounded in UB CSE data. The system follows the general RAG approach introduced by Lewis et al. [1] and summarized in later RAG survey work [2], but it adds a structured graph retrieval path because many advising questions are relational rather than purely textual. This design matches the broader view that large language models and knowledge graphs solve different parts of the knowledge access problem [3]. The system also draws inspiration from GraphRAG work [4], while remaining simpler and more application-specific: it uses an explicitly built `Neo4j` graph rather than an automatically inferred global community graph.

The goal of the project is not to replace academic advisors. The goal is to make public, factual departmental information easier to query. The system is therefore designed around grounding: every answer should be based on retrieved graph rows, retrieved document chunks, or both.

2 Project Goals

The project has four practical goals.

First, unify scattered departmental information. The system should answer over course records, faculty profiles, lab pages, research-area pages, and handbook content without requiring students to search each source separately.

Second, separate exact relationships from descriptive text. Questions such as "Who teaches CSE 435?" need structured retrieval, while questions such as "What does the DRONES Lab do?" need descriptive document retrieval. MyVictor uses both methods instead of forcing all questions into a single retrieval model.

Third, maintain traceable data pipelines. The data pipeline stores crawled files locally, builds a graph through repeatable scripts, and indexes documents through a dedicated vector indexer. This makes debugging and rebuilding possible.

Fourth, evaluate the system with repeatable tests. The repository includes a golden-question evaluation harness with deterministic checks and optional LLM judging.

3 High-Level Architecture

Figure 1 shows the core system architecture. The key idea is that the same user question is sent to two retrieval paths. The graph path converts the question into `Cypher` and queries `Neo4j`. The vector path embeds the question, searches `Qdrant`, and reranks retrieved chunks. The answer generator receives both contexts.

The implementation uses `Python`, `Flask`, `HTMX`, `Neo4j`, `Qdrant`, `LlamaIndex`, `Playwright`, and OpenAI-compatible model endpoints. `Qwen3` is used for answer generation and `Cypher` generation [5]. `BGE-M3` provides dense embeddings, and `BGE reranker v2 M3` is used for reranking retrieved chunks [8]. `BGE-M3` is a good fit for this retrieval setting because its paper and model card describe multilingual, multi-function, and long-context retrieval support [6, 7]. `Qdrant` and `LlamaIndex` both document support for hybrid retrieval that combines dense and sparse vectors [9, 10].

4 Corpus Inventory

Figure 2 summarizes the local data used by the inspected repository. These counts are from the files present in the project, not estimates.

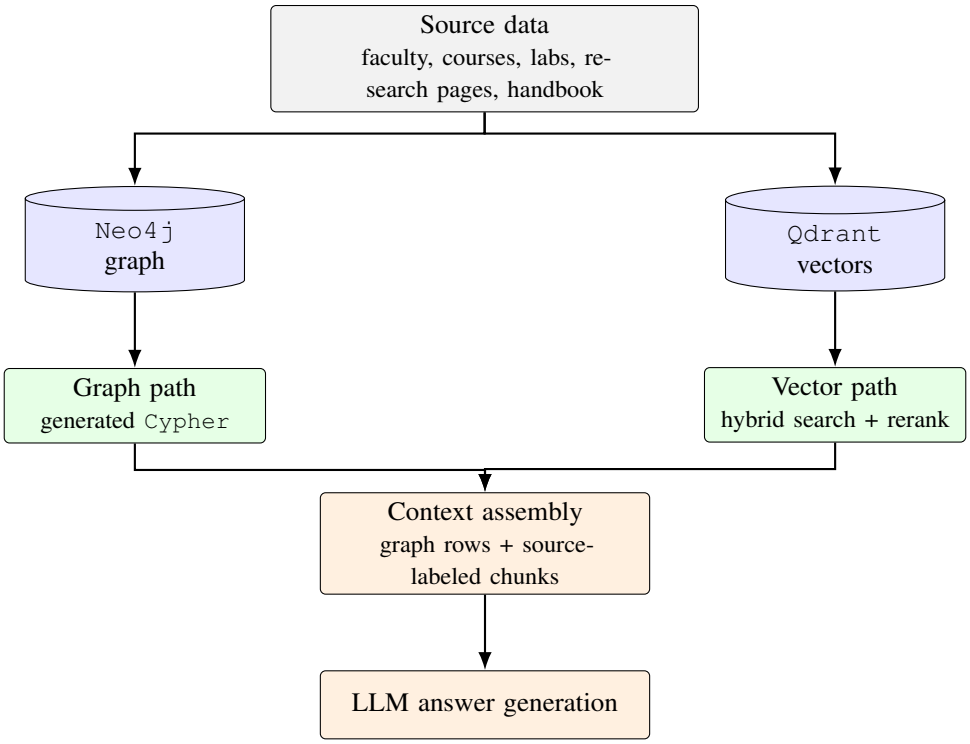

Figure 1: MyVictor architecture. The graph and vector stores are built separately, then queried together at runtime.

The largest raw source is crawled lab HTML. However, the most reliable entity-level data comes from the JSONL files. This is why the system uses both structured and unstructured ingestion: HTML provides breadth and descriptive content, while JSONL provides cleaner factual fields.

5 Data Acquisition Pipeline

The lab crawler is implemented with `Playwright`'s asynchronous `Python` API. `Playwright` supports browser automation across modern rendering engines and provides both synchronous and asynchronous `Python` APIs [12]. This is useful for departmental and lab pages because their structure is not uniform. Some pages are plain HTML, some are hosted on Buffalo.edu content systems, and some are external lab websites.

The crawler is designed to collect useful pages while avoiding runaway traversal. It applies allowed-domain checks, maps whitelisted external lab domains to canonical folder names, limits crawl depth, and enforces a per-lab page budget. It also filters URLs that are unlikely to contain useful page text, including login pages, CMS edit routes, media assets, archives, images, PDFs, Office documents, and JavaScript or mail links.

Same-lab scoping is one of the most important parts of the crawler. A lab can be represented by a dedicated subdomain, a folder path on a shared university domain, or a whitelisted external domain. The crawler uses these hosting patterns to decide whether a link belongs to the current lab. Without this boundary, one lab crawl could drift into unrelated department pages and pollute the dataset.

After download, the HTML-to-text cleaner removes scripts, styles, navigation, headers, footers, sidebars, iframes, and breadcrumbs. It also deduplicates repeated lines. The cleaner favors simple, readable extraction rules rather than page-specific logic. This makes the behavior easier to audit, but it also means that later

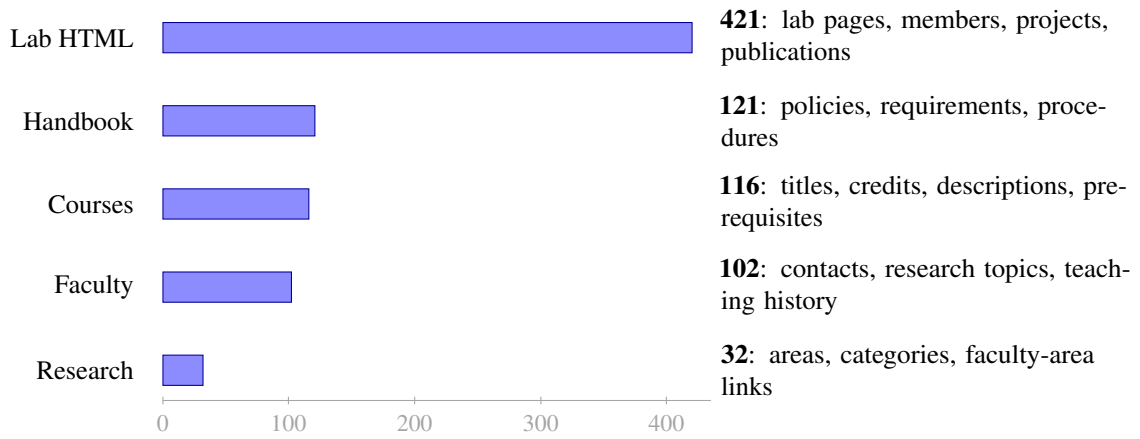

Figure 2: Corpus inventory and relative scale. Lab HTML pages are the largest unstructured source, while JSONL records provide structured faculty and course facts.

website redesigns may require extraction updates.

6 Knowledge Graph Construction

The `Neo4j` graph models the parts of the department that are naturally relational. `Neo4j` is queried through `Cypher`, a declarative graph query language documented by `Neo4j` [11]. The graph contains typed nodes and typed relationships.

6.1 Graph Entities

The main node labels are grouped around advising concepts:

- **Academic structure:** `Department`, `ResearchCategory`, `ResearchArea`, `Degree`, and `Policy`.
- **People:** `Faculty` and `LabMember`.
- **Teaching:** `Course`.
- **Research activity:** `ResearchTopic`, `Lab`, `Project`, `Publication`, and `Conference`.

These labels allow the system to answer questions over people, courses, labs, policies, and research themes without relying only on document similarity.

6.2 Graph Relationships

The most important relationships are `TEACHES`, `RESEARCHES`, `AFFILIATED_WITH`, `MEMBER_OF`, `WORKS_IN`, `HAS_PREREQUISITE`, `HAS_LAB`, `HAS_PROJECT`, `HAS_PUBLICATION`, and `OFFERS`. These relationships are what make the graph path valuable. For example, a course prerequisite query can follow `HAS_PREREQUISITE`; a faculty research query can follow `RESEARCHES`; and a lab membership query can follow membership edges.

6.3 Build Stages

The graph is built in three ordered stages.

Stage 1 builds research categories, research areas, seed faculty, topics, labs, and conferences. It reads Markdown files under the research-pages directory. This stage also normalizes lab names to prevent duplicate nodes when the same lab appears under different names or abbreviations.

Stage 2 enriches labs from crawled HTML. It extracts lab descriptions, lab members, projects, and publications from lab folders. It also attempts to connect extracted lab members back to existing faculty nodes.

Stage 3 loads structured faculty and course data. Faculty JSONL records enrich faculty nodes with email, office, education, biography, research topics, and teaching history. Course JSONL records create course nodes and prerequisite edges. Course codes are normalized from compact forms such as `CSE501` into the user-facing form `CSE 501`.

Each stage uses `MERGE` operations and uniqueness constraints so that rerunning a build stage updates existing nodes instead of creating duplicate nodes. This is important for a system that may be rebuilt after new crawls.

7 Vector Indexing and Hybrid Search

The vector indexer handles the document side of the system. It reads research pages, cleaned lab HTML, and handbook sections. The handbook is split into section-level documents, which is better than indexing it as one long text. For policy questions, section-level retrieval is more precise because each retrieved document has a coherent topic.

Documents are chunked with a sentence-aware splitter using 1024-token chunks and 200-token overlap. The overlap reduces the chance that a relevant explanation is split across two chunks with insufficient context. Metadata is attached to each chunk so retrieved text can be traced back to its source type, filename, lab folder, or handbook section.

The `Qdrant` collection uses hybrid retrieval: dense vectors from the embedding model and sparse BM25-style vectors from `FastEmbed`. This is useful because academic advising queries mix exact strings and semantic concepts. Course codes, names, emails, and terms benefit from sparse matching. Broader questions about research topics or policies benefit from dense semantic similarity.

The vector path retrieves more candidates than it finally needs, then calls a reranker. The reranker scores query–document pairs more directly than the initial embedding search. If the reranker endpoint fails, the code falls back to the original `Qdrant` retrieval order rather than failing the entire user request.

8 Runtime Query Flow

Figure 3 shows the query-time behavior. The key performance decision is parallel retrieval: graph and vector retrieval are dispatched together, then joined before answer generation.

8.1 Graph Path

The graph path uses three mechanisms. First, it checks an in-memory `Cypher` cache using a normalized version of the question. Second, it checks conservative regex shortcuts for simple frequent requests such as

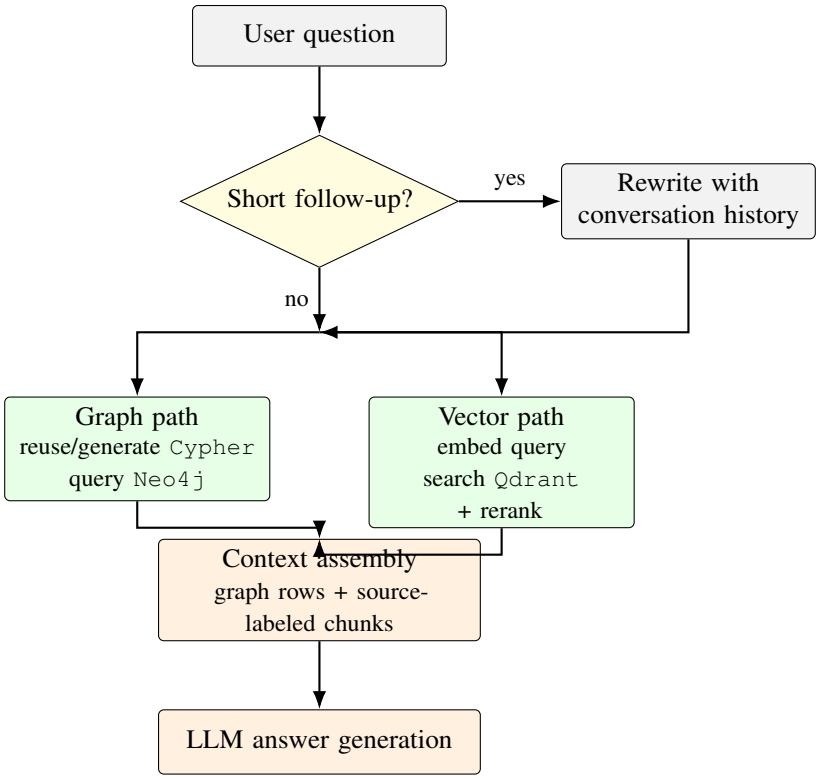

Figure 3: Runtime query flow. The graph and vector paths run concurrently, then the answer model receives both contexts.

listing all labs, faculty, courses, degrees, or research areas. Third, it calls the language model to generate `Cypher` for complex questions.

The shortcut layer is intentionally narrow. A question such as "list all labs" can be answered with a deterministic query. A question such as "which labs work on machine learning?" should not match a generic shortcut because it requires semantic filtering or graph traversal.

The Cypher-generation prompt includes rules learned from implementation bugs. Teaching terms are stored in title case, so semester filters must be case-insensitive. Lab affiliation can appear through multiple graph paths, so lab membership queries need to account for `MEMBER_OF`, `WORKS_IN`, and `LabMember` membership patterns. Name matching uses case-insensitive substring logic to handle natural user input.

The code uses the raw OpenAI-compatible client for `Cypher` generation so it can pass model-specific settings that disable `Qwen3`'s thinking mode. For this task, the desired output is a short valid `Cypher` query, not a long explanation.

8.2 Vector Path

The vector path embeds the question, retrieves candidates from `Qdrant`, reranks them, and formats the top chunks. Each chunk is labeled with source metadata. These source labels help the answer prompt distinguish where information came from, even though the current interface does not expose formal citations to users.

8.3 Answer Generation

The final prompt receives recent conversation history, graph results, and vector results. It instructs the model to trust graph rows for exact relational facts and use vector chunks for descriptive context. This division is important. Graph rows are better for lists, counts, prerequisites, and explicit relationships. Vector chunks are better for descriptions of labs, research areas, and handbook policies.

The `Flask` web route is asynchronous. `Flask` supports `async def` views when installed with async support, and its documentation notes that async views are helpful for concurrent I/O-bound work even though each request still occupies a worker [13]. MyVictor's retrieval workload is I/O-heavy, so this model is appropriate for the current application.

9 Web Interface

The web application uses `Flask` templates and `HTMX` partial rendering. The user submits a message through the chat form. The server appends the user message, calls the chatbot, converts the model response from Markdown to HTML, and returns the updated message list as a partial template. The browser updates the chat area without a full page reload.

The JavaScript layer handles textarea resizing, enter-to-send behavior, optimistic display of the user's message, hiding the typing indicator after a response, and scrolling the chat window to the bottom.

One implementation limitation is that the web module stores the chatbot instance and messages as module-level globals. This is acceptable for a demo or single-user local run, but it is not a production session model. Multiple users would share one message history. A production deployment should store history per session or per user and should manage model and database clients through explicit application lifecycle hooks.

10 Evaluation Harness

The repository includes an evaluation runner under `graph_rag/eval`. The golden dataset is JSONL. Each item contains an identifier, question, category, difficulty, expected retrieval path, required mentions, forbidden mentions, expected facts, and tags.

The runner resets chatbot history for each item, calls the chatbot, records latency, stores the generated `Cypher` query, and applies deterministic string checks. Required mention checks are useful for exact facts such as course codes, faculty names, and email addresses. Forbidden mention checks catch obvious hallucinations or incorrect entities.

The evaluation code can also call an LLM judge. The judge scores faithfulness, correctness, and unjustified refusal behavior. This is useful as a secondary signal, but it should not be treated as a perfect oracle. In the inspected implementation, the judge uses the same model family as the chatbot, which can create self-preference bias. A stronger evaluation setup would use an independent judge model and add component-level metrics such as `Cypher` validity and `retrieval recall@k`.

11 Implementation Scope

This paper describes the functionality that is present in the inspected repository. The current implementation includes data ingestion, graph construction, vector indexing, hybrid retrieval, web chat, and an evaluation harness. It does not yet include every feature that would be expected in a hardened production deployment.

In particular, the inspected code does not contain a completed backend response-safety interceptor. The `Flask` route calls the chatbot, converts the answer to HTML, and returns the rendered message list, but it does not run a separate safety classifier or guardrail before returning the answer. A response-safety layer is still useful future work, especially for deployment, but it should be implemented, logged, and tested before being described as part of the working system.

All corpus counts in this paper are taken from the local repository state: 421 lab HTML files, 32 research Markdown files, 121 handbook sections, 102 faculty records, and 116 course records. Using repository-derived counts keeps the report reproducible and avoids relying on stale numbers from earlier development notes.

12 Limitations

Data freshness. The system answers from a snapshot. If UB CSE websites, catalog pages, or handbook content change, the crawler, graph builders, and vector indexer must be rerun.

HTML extraction fragility. Lab pages are heterogeneous. A website redesign can silently reduce extraction quality even if the crawler still downloads pages successfully.

Session isolation. The current `Flask` implementation uses shared module-level message state. This should be replaced before multi-user deployment.

Incomplete safety layer. The inspected code does not include a completed response-safety interceptor. Guardrails should be added before making safety claims.

Operational dependencies. A complete run requires valid model API credentials, `Neo4j`, `Qdrant`, and populated stores. Local `Qdrant` file mode is helpful for development, but stale lock files can become an operational nuisance after interrupted runs.

13 Future Work

The next engineering steps should focus on reliability and correctness.

First, the web app should move to per-session or per-user conversation state. Second, the data refresh process should be scheduled and auditable. Third, the eval harness should add component-level metrics such as generated-`Cypher` validity, graph result correctness, and retrieval recall. Fourth, lab extraction should be audited after each crawl so that page-structure changes do not silently degrade the graph. Fifth, a real response-safety guardrail should be integrated into the backend response path and covered by tests. Finally, the LLM judge should be replaced with an independent model to reduce self-preference bias.

14 Conclusion

MyVictor demonstrates a practical way to combine graph retrieval and vector retrieval for academic advising. The graph path gives precise answers for structured relations such as teaching assignments, prerequisites, lab affiliations, and faculty research topics. The vector path retrieves descriptive evidence from lab pages, research pages, and handbook sections. Running both paths in parallel gives the language model complementary evidence while keeping latency manageable.

The main lesson is that advising RAG systems need more than a vector database. They need structured institutional relationships, careful document cleaning, source-aware retrieval, conservative query-generation

rules, and repeatable evaluation. MyVictor provides that foundation for UB CSE, while leaving clear next steps for production hardening.

References

[1] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel, S. Riedel, and D. Kiela, "Retrieval-augmented generation for knowledge-intensive NLP tasks," in *Advances in Neural Information Processing Systems*, vol. 33, pp. 9459–9474, 2020.

[2] Y. Gao, Y. Xiong, X. Gao, K. Jia, J. Pan, Y. Bi, Y. Dai, J. Sun, M. Wang, and H. Wang, "Retrieval-augmented generation for large language models: A survey," arXiv:2312.10997, 2023.

[3] S. Pan, L. Luo, Y. Wang, C. Chen, J. Wang, and X. Wu, "Unifying large language models and knowledge graphs: A roadmap," *IEEE Transactions on Knowledge and Data Engineering*, vol. 36, no. 7, pp. 3580–3599, 2024, doi: 10.1109/TKDE.2024.3352100.

[4] D. Edge, H. Trinh, N. Cheng, J. Bradley, A. Chao, A. Mody, S. Truitt, and J. Larson, "From local to global: A Graph RAG approach to query-focused summarization," arXiv:2404.16130, 2024.

[5] Qwen Team, "Qwen3 technical report," arXiv:2505.09388, 2025.

[6] J. Chen, S. Xiao, P. Zhang, K. Luo, D. Lian, and Z. Liu, "M3-Embedding: Multi-linguality, multi-functionality, multi-granularity text embeddings through self-knowledge distillation," arXiv:2402.03216, 2024.

[7] BAAI, "BAAI/bge-m3 model card," Hugging Face, accessed May 2026. Available: https://huggingface.co/BAAI/bge-m3.

[8] BAAI, "BAAI/bge-reranker-v2-m3 model card," Hugging Face, accessed May 2026. Available: https://huggingface.co/BAAI/bge-reranker-v2-m3.

[9] Qdrant, "Qdrant documentation," accessed May 2026. Available: https://qdrant.tech/documentation/.

[10] LlamaIndex, "Qdrant hybrid search," accessed May 2026. Available: https://developers.llamaindex.ai/python/framework/integrations/vector_stores/qdrant_hybrid/.

[11] Neo4j, "Neo4j documentation," accessed May 2026. Available: https://neo4j.com/docs/.

[12] Microsoft, "Playwright for Python documentation," accessed May 2026. Available: https://playwright.dev/python/docs/intro.

[13] Pallets, "Using async and await," Flask Documentation, accessed May 2026. Available: https://flask.palletsprojects.com/en/stable/async-await/.