Mixed-Method (Cache and Retrieval) Augmented Generation System for Northeastern's Office of Global Studies - Experimentation Phase

David Johnson
Himanshu Tahelyani

Northeastern University

DS 5500 Capstone: Application in Data Science

Spring 2025

March 20, 2025

Abstract

Navigating complex university websites for essential information can be challenging for international students seeking guidance on visas, applications, and work opportunities. To help, we develop a Mixed-Method Augmented Generation system combining Retrieval-Augmented Generation (RAG) and Cache-Augmented Generation (CAG) to enhance response accuracy and efficiency. The system utilizes web-scraped content from Northeastern University's Office of Global Services (OGS) website, with RAG retrieving relevant documents using Snowflake embeddings and a Cross-Encoder reranker, while CAG precomputes KV-caches to reduce response time for frequently asked queries.

Our experimentation phase evaluated retrieval sizes, response accuracy, and fallback validation methods, revealing a trade-off between speed and accuracy. While CAG significantly reduced inference time, while model hallucination and assumption-based responses posed challenges. Future improvements will focus on refining document structuring, optimizing query routing, and mitigating hallucinations for scalable real-world deployment.

Contents

Chapter 1

Introduction and Related Work

1.1 Introduction

The experimentation phase plays a critical role in optimizing the system by quantitatively and qualitatively evaluating model performance and output quality. Given the goals of providing quick and accurate content from the OGS website, evaluating these items is critical. This phase investigates whether each system (CAG and RAG) provide high-quality, accurate responses which answer user queries. Through a mix of automated and manual testing, we refine the components that make up each of the two main augmented generation methods.

1.2 Related Work

Prior research in cache-augmented generation has demonstrated efficiency in generating accurate answers to queries against a data set pre-tokenized and loaded to a models cache. More specifically, studies on KV-cache optimizations in models such as Llama 3.1 have highlighted their ability to reduce token recomputation overhead. Importantly, this works best on data sets of a limited size which could fit inside a models context window. As discussed in [Cha+25], Cache-Augmented Generation can replace RAG for certain tasks.

Chapter 2

Objectives and Hypotheses

2.1 Objectives

The objectives with the CAG experimentation are to ensure the model can generate answers that are accurate and based solely on the documents loaded into the models context window, and faster than the using RAG to retrieve documents dynamically. Another important and related goal is to ensure the model outputs consistently identifiable content that would indicate the model cannot answer the query given the preloaded context, so that the system can appropriately route the query to the RAG function.

2.2 Hypotheses

- Implementing CAG will significantly reduce the inference time for frequently asked queries.
- Implementing RAG will retrieve more nuanced and relevant information when the CAG context is insufficient.
- A hybrid CAG-RAG approach will outperform a standalone RAG in response efficiency without sacrificing accuracy.

Chapter 3

Experimental Setup

3.1 Hardware and Software (CAG)

Our experiments were carried out on a system equipped with an NVIDIA RTX 3070 GPU featuring 8GB of GDDR6 memory, paired with a modern 13th-gen Intel i7-13700K. We used Python as our primary programming language along with PyTorch as the deep learning framework. The Hugging Face Transformers library [Fac25] was necessary for model loading and inference, while BitsAndBytes was utilized to enable efficient 4-bit quantization, reducing memory overhead. Additional libraries such as SentenceTransformers, evaluate, and standard Python modules (e.g., gc, time, and json) were employed to handle tasks ranging from embedding computation to resource management and data serialization.

3.2 Model Selection (CAG)

In the model selection phase, we considered several models, including meta-llama/Llama-3.1-8B-Instruct, microsoft/Phi-3.5-mini-instruct, and mosaicml/mpt-7b-8k-instruct. Meta-llama/Llama-3.1-8B-Instruct, with its 8-billion parameters and 128k context window, demonstrated strong conversational capabilities and nuanced response generation but required significant GPU memory, which could be challenging on an 8GB card. MosaicML/mpt-7b-8k-instruct offered an 8k context window and balanced performance, yet its resource demands remained relatively high given its 7-billion parameters while having the smallest context window of our evaluated models. Ultimately, we finalized Microsoft/Phi-3.5-mini-instruct as our model of choice. With roughly 3.82 billion parameters and an impressive 128k context length, this model strikes an optimal balance between efficiency and capacity. Its instruction tuning ensures that responses are closely aligned with the input context, which is a critical requirement for our cache-augmented generation tasks, while its compact size allows it to run effectively on our available hardware without necessitating overly

aggressive quantization strategies.

Chapter 4

Dataset and Preprocessing

4.1 Dataset

The dataset for this study was collected through web scraping from the Northeastern University Office of Global Services (OGS) website. Using Scrapy [Scr25] and BeautifulSoup [Bea25], the scraper systematically extracted text from multiple pages, including the homepage, informational pages on visa applications, employment authorization, and student services. The scraper followed internal links using an allowed domain to give a finite range to ensure comprehensive coverage of relevant content. The dataset consists of structured JSON-formatted records, where each entry includes:

- URL of the scraped page
- Title of the webpage
- Full page content extracted from the *content* section
- Sections list, capturing structured information by extracting text from section elements, along with associated headers

Given the nature of web-scraped content, inconsistencies in formatting and missing section headers were observed, requiring additional cleaning.

4.2 Preprocessing

Several preprocessing steps were applied to enhance data usability:

- Text Cleaning: Extraneous whitespace was removed, and text was normalized with period-space delimiters to preserve sentence structure.
- Section Structuring: Headers were extracted where available, and missing headers were replaced with "No Header found" placeholders.

- JSON Formatting Fixes: To prevent duplicate or malformed JSON outputs, the Scrapy pipeline was adjusted to ensure a single well-formed list of records.
- Tokenization Considerations: Since this dataset is intended for language model processing where context is critical, stop word removal and lowercasing were deliberately omitted to preserve contextual integrity, following best practices in transformer-based NLP models.

In the end, we used just the sections portions of each page which gave us a simple way to parse the content in each page. We stored the section and its related URL in our data-store for the RAG portion of the system so the URL could be retrieved when RAG was used, and discarded the rest.

Chapter 5

Training and Validation Process and Metrics

5.1 CAG - Model and Cache Initialization

Our approach leverages recent pre-trained language models rather than training from scratch. In this work, we utilize the Microsoft Phi-3.5-mini-instruct model as the generative core for producing responses to test questions. Rather than employing any fine-tuning, our process centers on robust model deployment and iterative testing. To accommodate limited GPU memory and avoid runtime errors, we implemented a dynamic memory management strategy, including:

- **Memory Management and Retry Mechanism** A custom function `clear_cuda_memory` ensures that GPU memory is released after each iteration by synchronizing operations, collecting garbage, and emptying the CUDA cache. Additionally, the model-loading routine attempts to load the model multiple times (up to 10 retries) with periodic memory cleanups to handle `OutOfMemoryError` exceptions gracefully.
- **Cache Generation** We further extend the model's capabilities by generating a key-value cache using an input system prompt derived from a set of n documents. This cache serves as a basis for guiding the model's response, allowing us to quickly retrieve relevant document fragments during inference.

While conventional training parameters such as batch sizes and epochs are less applicable in this context, our iterative testing, executed over different subsets of the document corpus (e.g., n= 5, 10, and 20 documents per test iteration), ensures that the deployed model performs reliably under various conditions.

5.2 CAG - Validation Strategy and Metrics

Our validation strategy is designed to assess the quality of generated answers using multiple modern evaluation metrics. These metrics target different aspects of answer quality, including syntactic correctness and semantic relevance:

- **Token-Level F1 Score:** Both the generated answer and the true answer are normalized (e.g., lowercasing and stripping whitespace), and token overlap is computed. Precision, recall, and the resulting F1 score provide an initial gauge of textual similarity.
- **Embedding-Based Semantic Similarity:** Recognizing the limitations of syntactic comparison, we employ dense vector representations using Sentence-BERT (specifically, the `all-MiniLM-L6-v2 model`). The generated and true answers are converted into embeddings, mean-pooled, and compared using cosine similarity. This method is more robust to paraphrasing and captures the underlying semantic content.
- **Zero-Shot Natural Language Inference (NLI):** A zero-shot NLI pipeline (using the `cross-encoder/nli-roberta-base model`) is applied to determine whether the generated answer logically entails the true answer. This evaluation step helps ensure that the response not only resembles the target text but also conveys the correct information.
- **Text Classification for Relevance:** As a proxy for overall answer relevance, a text classification pipeline (with `distilbert-base-uncased-finetuned-sst-2-english`) provides an additional score, indicating the sentiment or relevance of the concatenated generated and true answers.

In cases where the primary evaluation does not meet a preset quality threshold (e.g., scores below 0.7), a secondary validation process is triggered. This secondary process involved **Claim Extraction and Semantic Matching** where the generated answer is segmented into individual claims using sentence splitting. For each claim, we retrieve candidate passages from the document corpus based on semantic similarity (computed via `Sentence-BERT`). A claim is marked as supported if any candidate passage exceeds a defined similarity threshold. The overall validation decision is based on the aggregate ratio of supported claims.

Hyperparameters such as similarity thresholds (0.35 for candidate retrieval and 0.45 for claim verification) and aggregate thresholds (0.8 for overall acceptance) were established based on preliminary experiments. Although automated tuning methods like grid search or Bayesian optimization were not explicitly implemented, our iterative testing and adaptive retry mechanisms provided a form of continuous process refinement during the evaluation process.

5.3 CAG - Final Testing

Final testing was performed on a set of unseen document-question-answer (QA) pairs to ensure reproducibility and generalization of our results. The QA pairs were generated by feeding the individual documents one at a time to OpenAIs gpt-4o-mini via their API and requesting 4 questions-answers, 3 that are answerable and 1 not answerable (but related), for each document The testing framework then follows these steps:

5.3.1 Random Sampling of Test Data

A helper function (`select_random_n`) uniform-randomly selects a subset of n documents and their associated test QAs from the larger dataset to respond to a set of n questions. This sampling process, executed over multiple iterations, ensures that our evaluation covers diverse scenarios.

5.3.2 Iterative Evaluation Loop

For each test iteration, the following steps are executed:

- **Model Preparation:** The system loads the pre-trained model with error handling and cache generation based on the selected documents.
- **Query Response Generation:** A random question from the test QA set is posed to the model, which generates a corresponding answer.
- **Multi-Metric Evaluation:** The generated answer is first evaluated against the true answer using the primary metrics (token-level F1, embedding similarity, zero-shot NLI, and text classification). If these metrics indicate insufficient quality, a secondary validation via claim-based evidence retrieval is performed.
- **Result Aggregation and Logging:** The results from each iteration (including evaluation scores and pass/fail outcomes) are saved into JSON files for further analysis.

5.3.3 Reproducibility and Transparency

Each testing iteration is accompanied by detailed logging of memory usage, retry attempts, and evaluation metrics. This transparency facilitates reproducibility and helps troubleshoot potential issues in the evaluation pipeline.

By integrating these steps, our final testing process ensures that the deployed model is validated on unseen and reliable QA data and that the evaluation metrics capture both syntactic and semantic aspects of answer quality.

Chapter 6

Results

6.1 CAG Quantitative Results

The initial pass at testing with our QA set did 5 rounds for each set of N documents where N= 5, 10, and 20. Initially the secondary validation used when the True vs Generated answer could not find a match involved a similarity search between the answer and the entire set of N documents, which resulted in all generated answers being marked a True, or matching the information in the document set. This validation was replaced with the *Claim Extraction and Semantic Matching*, which has thus far struggled to complete a larger round of testing due to repeatedly running into `OutOfMemoryError` on my machine.

Our results without the secondary test are as follows:

- 5-document tests: 72
- 10-document tests: 68
- 20-document tests: 70.5

This shows reasonable accuracy when using these automated methods, but not as high as would be preferred for our experiment and necessary for a production system.

6.2 CAG Qualitative Analysis

Manually reviewing the generated answers for their accuracy showed that the model often includes suggestions and context that did not directly come from the text, but was not always inaccurate. Usually telling the user that they should contact OGS or other NU offices for further information. Occasionally, the model would assume information based on the document context, which may not be preferred for a production system. Typically tweaking the model prompt to explicitly instruct the model to include explicitly document-sourced information decreased these types of assumptions, but not completely.

6.3 Interpretation

The key finding revolves around model hallucination and generated assumption, as mentioned in the previous section. It was difficult to get the model to reliably output only information included in the text, or get it to output specific and limited characters when the information could not be found so that the query could be routed to the RAG portion of the system. 'Not Found' outputs would usually contain information on who a user might contact to get the information they need, or provide an assumption clearly based on the models pre-training. However, when the model was accurate, it was very informative and quick. Inference time after the initial KV Cache load which took about 5 seconds, was usually under 2 seconds. So while work needs to be done on accuracy, we were able to achieve the objective of fast inference time.

Chapter 7

Discussion and Iterative Improvements

7.1 CAG Model Adjustments

The primary generation models was not fine-tuned, and the only custom portion of the architecture was pre-tokenization and iterative management of the context, as well as using a custom greedy generation function which followed [Met24]. These changes enabled the functionality of CAG, where the usual pipeline functions would have been insufficient for cache augmented generation.

7.2 Data Adjustments

No significant customization was performed around collecting and feeding the data to our models. Once the pages were broken into sections and HTML tags were removed keeping on the text, those were added to a list which we used as-is.

7.3 Future Work

Future work for better data collection would be to use an embedding based approach to iterated through each ¡P¿ tag of each page section, then comparing the subsequent P to the prior using an embedding model, and if sequential portions are determined to be similar, then they are considered one document, otherwise parse into separate documents. How to handle model routing and hallucinations could also be improved. For example we could implement an agent model which attempts to route the query to the RAG or CAG side based on some search function. We could also fine-tune our generative model to understand how to respond better when a question cannot be answered with more concrete

and consistent responses rather than our current results which often include assumptions and other information the model assumed would be helpful.

Chapter 8

Conclusion

The experimentation phase provided important insights into the effectiveness of Cache-Augmented Generation (CAG) and Retrieval-Augmented Generation (RAG) in optimizing response accuracy and efficiency for our Northeastern University Office of Global Studies chatbot. Our findings confirmed that CAG significantly reduces response latency, achieving inference times under two seconds after initial KV-cache loading, while RAG ensures high-quality responses for complex queries using advanced embeddings and cross-encoder reranking. However, challenges in model hallucination and inaccurate response generation highlight areas requiring further improvement.

These results directly contribute to the broader project objectives by demonstrating that a hybrid CAG-RAG approach has the potential to effectively balance speed and accuracy. The system individual components retrieves and processes OGS information, improving accessibility for international students seeking visa, application, and job-related guidance. However, further refinements are necessary to enhance response reliability, particularly in preventing overgeneralization and ensuring accurate fallback mechanisms for when questions cannot be answered.

Future work will focus on refining document structuring techniques during preprocessing, optimizing query routing between CAG and RAG through an agent-based decision system, and exploring fine-tuning approaches to mitigate model hallucination. Additionally, embedding-based document segmentation strategies will be explored to improve document content homogeneity. These improvements will further strengthen the system's scalability and reliability for real-world deployment.

Bibliography

[Met24] Sabay Bio Metzger. *Cache-Augmented Generation (CAG) from Scratch*. [Online; accessed 15-Feb-2025]. Feb. 2024. URL: https://medium.com/@sabaybiometzger/cache-augmented-generation-cag-from-scratch-441adf71c6a3.

[Bea25] BeautifulSoup. *BeautifulSoup Documentation*. [Online; last accessed 16-Mar-2025]. Mar. 2025. URL: https://www.crummy.com/software/BeautifulSoup/bs4/doc/.

[Cha+25] Brian J. Chan et al. "Don't do RAG: When Cache-Augmented Generation is All You Need for Knowledge Tasks". In: *arXiv* abs/2412.15605 (Feb. 2025). URL: https://arxiv.org/abs/2412.15605.

[Fac25] Hugging Face. *Transformers and Model Hub*. [Online; last accessed 16-Mar-2025]. Mar. 2025. URL: https://huggingface.co.

[Scr25] Scrapy. *Scrapy Web Crawling Framework*. [Online; last accessed 16-Mar-2025]. Mar. 2025. URL: https://scrapy.org.