Implementing Retrieval Augmented Generation Technique on Unstructured and Structured Data Sources in a Call Center of a Large Financial Institution

Syed Shariyar Murtaza[1], **Yifan Nie**[1], **Elias Avan**[1], **Utkarsh Soni**[1], **Wanyu Liao**[1], **Adam Carnegie**[2], **Cyril John Mathias**[2], **Junlin Jiang**[2] **and Eugene Wen**[1]

[1]Manulife, 200 Bloor St E, Toronto, ON M4W 1E5, Canada
[2]John Hancock , 200 Berkeley St, MA 02116, USA
[1]{syed_shariyar_murtaza,yifan_nie,elias_abdollahnejad}@manulife.com
[1]{utkarsh_soni,vanessa_liao,eugene_wen}@manulife.com
[2] {acarnegie,cyril_mathias,junlin_jiang}@jhancock.com

Abstract

The retrieval-augmented generation (RAG) technique enables generative AI models to extract accurate facts from external unstructured data sources. For structured data, RAG is further augmented by function calls to query databases. This paper presents an industrial case study that implements RAG in a large financial institution's call center. The study showcases experiences and architecture for a scalable RAG deployment. It also introduces enhancements to RAG for retrieving facts from structured data sources using data embeddings, achieving low latency and high reliability. Our optimized production application demonstrates an average response time of only 7.33 seconds. Additionally, the paper compares various open-source and closed-source models for answer generation in an industrial context.

1 Introduction

With the rapid development of Generative AI technologies (et al., 2020), the retrieval-augmented generation (RAG) (Chen et al., 2024; Zhang et al., 2024) technique has become popular in academia and industrial applications (Zhu et al., 2024; Lashinin et al., 2023; Shahin et al., 2024). RAG involves two phases: ingestion, where document chunks are vectorized and stored in vector databases, and inference, where relevant chunks are retrieved to answer questions using a Large Language Model (LLM). Although RAG is effective with unstructured data, industrial applications often involve structured data. A common approach in the literature to retrieve structured data is to leverage LLM to translate a text query into a database-specific query (such as SQL), then call a database function to retrieve relevant facts (LangChain, 2024b,a). This approach increases the number of calls to LLM (incurring cost and delay) and sometimes it doesn't translate queries correctly.

In this paper, we present a case study on applying the RAG technique to a call center of a business unit of a very large financial institution. The call center has been in business for many decades. Its data span various structured and unstructured sources. When a customer calls, a customer service representative (CSR) answers the questions by looking up information from unstructured policy documents or structured data sources. Some of these sources can overlap and complicate the efforts of a CSR to respond to queries promptly. Our RAG application converts structured and unstructured data into chunks and vectorizes them using embedding models during the ingestion phase. This optimization improves latency (fewer LLM calls) and accuracy at inference time.

We implement our approach by converting headers and rows of structured data (database tables) into JSON strings and grouping them by business concepts. These JSON chunks are transformed into embeddings and stored in a vector database index. Similarly, we convert unstructured policy documents into chunks and store them in a separate index. During inference, we retrieve the top k relevant chunks from both indexes based on the input query, combine them into a prompt, and use GPT-3.5 to generate a grounded answer. An independent model (Llama 3 or GPT-4) validates the answer's quality with a confidence rating. We monitor performance by capturing confidence ratings, human feedback and response times.

Our production application has consistently generated accurate, grounded answers without hallucinations since May 2024. We observed occasional errors due to missing data or ambiguous contexts. These were fixed through updates to data pipelines and prompt revisions. We also optimized response times from an initial launch of an average of 21.91s to an average of 7.33 seconds. We present a comparison study of popular LLMs in the RAG application to facilitate model selection. Finally, we also present our application architecture, which will help the community in developing industrial-scale RAG applications.

2 Background and Related Work

To develop a RAG (retrieval-augmented generation) application, documents are first divided into smaller chunks (Finardi et al., 2024). These chunks can be created using a sliding window approach with some overlaps of words between chunks (Zhong et al., 2024), or through advance methods such as semantic chunking to keep semantically coherent text together in one chunk (Qu et al., 2024). Later, each chunk is indexed with its corresponding vector representation using an embedding model. During inference, these chunks are retrieved based on their semantic similarity with a question and are passed as part of the prompt to an LLM to generate an answer (Monir et al., 2024). If data is in a structured format like a relational database, then below are some of the methods to process the data for a RAG application.

Raw SQL Query: SQL is widely used for querying structured data due to its rapid query processing capabilities for real-time data analysis and simple syntax for SQL queries (Balkesen et al., 2018). (Faroult and Robson, 2006). SQL queries can be used to retrieve structured data in the RAG technique, and then LLM can generate the answer using the prompt created from the retrieved data. However, the raw SQL query approach could not be directly applied with a user's natural language query. The Text-to-SQL method is proposed to bridge this gap.

Text-to-SQL: To bridge the gap between natural language queries and SQL queries, the Text-to-SQL (Qin et al., 2022) approach converts natural language queries into SQL using encoder-decoder models, typically based on LSTM (Yu et al., 2018; Stower and Krechel, 2019) or Transformer architectures (Hwang et al., 2019; Lei et al., 2020). The encoder transforms natural language into vectors, while the decoder generates SQL queries, either through sketch-based methods (breaking down SQL clauses) or end-to-end generation (producing entire SQL queries). Text-to-SQL systems are user-friendly, eliminating the need for programming skills (Ahkouk et al., 2021). Modern Large Language Models (LLMs) can convert text to SQL. This means that we can use an LLM to convert a query to SQL in the RAG technique and then make a function call to a database to retrieve data (LangChain, 2024c). However, these models can be sensitive to input variations and may struggle with queries outside their training domain (Qin et al., 2022). This approach also increases the number of calls to an LLM, resulting in increased latency.

Training table embedding model: Other approaches such as TaPas (Herzig et al., 2020) use transformer-based architectures to pretrain tabular embedding models by flattening tables into 1-D sequences and adding various positional embeddings to understand table structures. The pretraining employs a masked language model loss function (Devlin et al., 2019), followed by fine-tuning with questions, tables, and answers. However, this method has limitations: it requires the full table input, which is impractical for large tables, and often only a subset of the table is relevant to the query, leading to noise and confusion.

3 Methodology

The architecture of our application is shown in Figure 1, with four major components: ingestion and indexing, inference, monitoring, and user interface.

3.1 Ingestion and Indexing

We collaborated with business partners to consolidate the data into three main sources: (a) general insurance policy documents for US states, (b) CSR notes, and (c) a structured database with specific customer policy information. Those sources are shown on the top right of Figure 1. Policy documents and CSR notes are stored in PDFs on Microsoft SharePoint and ingested into Azure Data Lake Storage (ADLS) upon updates, while structured data are ingested daily into Azure Synapse Lake for big data analysis. To implement the RAG technique for efficient answer generation, we vectorized (Karpukhin et al., 2020) both structured and unstructured data. Vectorization helps retrieve semantically relevant information more precisely

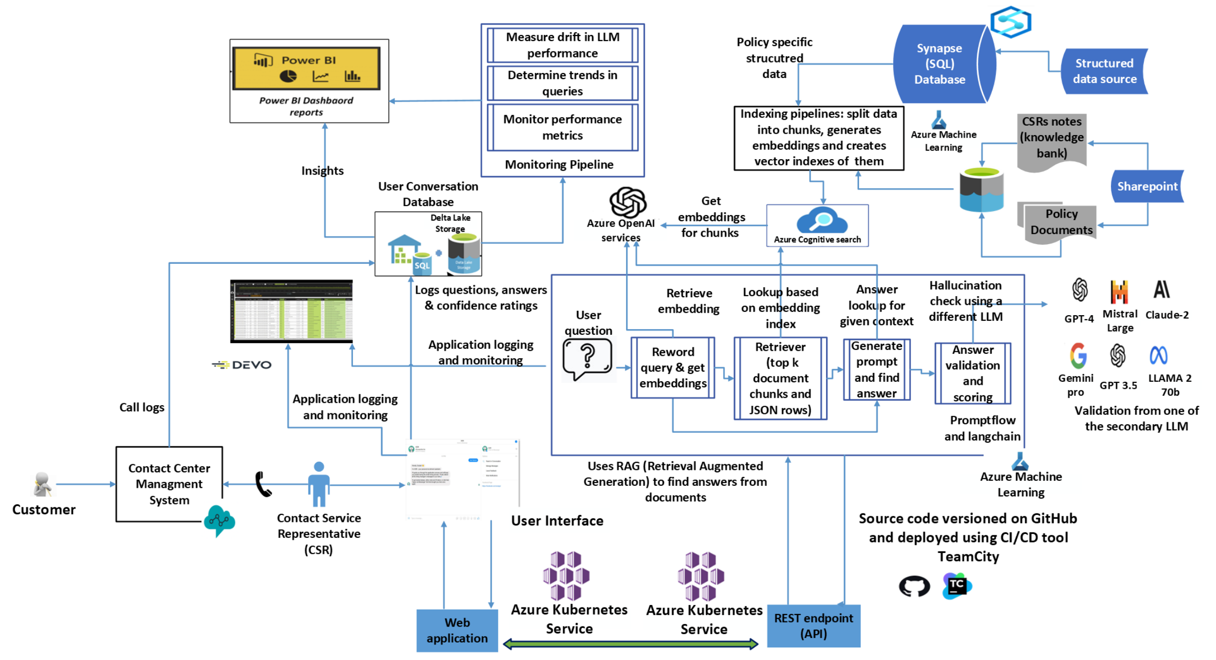

Power BI
Power BI Dashboard reports
Measure drift in LLM performance
Determine trends in queries
Monitor performance metrics
Monitoring Pipeline
Policy specific strucutred data
Synapse (SQL) Database
Structured data source
Indexing pipelines: split data into chunks, generates embeddings and creates vector indexes of them
Azure Machine Learning
CSRs notes (knowledge bank)
Sharepoint
Policy Documents
Insights
User Conversation Database
Delta Lake Storage
SQL
Azure OpenAI services
Get embeddings for chunks
Azure Cognitive search
Logs questions, answers & confidence ratings
Application logging and monitoring
DEVO
Call logs
Application logging and monitoring
User question
Retrieve embedding
Lookup based on embedding index
Answer lookup for given context
Hallucination check using a different LLM
Reword query & get embeddings
Retriever (top k document chunks and JSON rows)
Generate prompt and find answer
Answer validation and scoring
Promptflow and langchain
GPT-4
Mistral Large
Claude-2
Gemini pro
GPT 3.5
LLAMA 2 70b
Validation from one of the secondary LLM
Customer
Contact Center Managment System
Contact Service Representative (CSR)
Uses RAG (Retrieval Augmented Generation) to find answers from documents
Azure Machine Learning
User Interface
Source code versioned on GitHub and deployed using CI/CD tool TeamCity
Web application
Azure Kubernetes Service
Azure Kubernetes Service
REST endpoint (API)

Figure 1: Application Architecture

than a keyword search, particularly for structured data. It also facilitates in keeping latency low at inference time.

We indexed unstructured data (PDF files) by chunking text into 400-token segments with overlaps and vectorizing into 1536-dimensional vectors using the text-embedding-ada-002 model[1].These vectors are subsequently stored in an Azure AI Search [2] index using AI Search SDK.

Our structured data consists of large database tables that contain detailed information about each policy and client. These tables contain numerical, categorical, and textual information. An illustrative example is shown in Table 1. We implemented an innovative method to index structured data. Specifically, we de-normalized multiple tables in our structured database and also aggregated them by concepts; e.g., 'Comfort Keepers', 'Care Champions', etc. There were three distinct tables after our processing. There were 4.5 million rows in these tables after our processing compared to 50 million rows before processing them. Each row of each table is then converted into a JSON string with table headers as keys and cells as values. This is also shown in Table 1[3]. We used this JSON string as a chunk for vectorization. The maximum length of the JSON string (chunk) was 1300 tokens.

Table 1: Sample Tabular Data

Policy Number	Section Name	Product Rule	Benefit Amount
0000-0001	Policy Feature	ABC...	123.45
0000-0007	Policy Feature	DEF...	345.67

JSON representation for row 1: {'**Policy Number**':'0000-0001', '**Section Name**': 'Policy Feature' , '**Product Rule**':'ABC...', '**Benefit Amount**': '123.45' }

The JSON strings from all tables are vectorized using the text-embedding-ada-002 model and stored in one Azure AI Search index. This index was separate from the unstructured index. We also store metadata, such as policy numbers, state, city, page numbers, and file locations, with each JSON string. This meta-information facilitates precise and relevant information retrieval for queries (e.g., retrieving chunks relevant only to questions related to a specific policy number). It also provides references to sources (i.e., file locations) for validation of answers during inference.

The customer-specific policy values are updated regularly in the structured database. It is inefficient to re-index the entire dataset in AI Search database. We run a nightly job that detects updated policy numbers, indexes new records, and replaces existing vectors with updated ones. The new records are inserted into the existing index along with vectors.

[1]https://learn.microsoft.com/en-us/azure/ai-services/openai/concepts/models

[2]https://azure.microsoft.com/en-ca/products/ai-services/ai-search

[3]This is an illustrative table with synthetic data to show how the structured data are indexed, real data has more fields and due to its confidentiality, not presented here.

In addition, we optimize the speed of indexing by using parallelization in code and a higher throughput tier of Azure AI Search.

3.2 Inference

Inference is an important part of our implementation. We developed the inference application with Promptflow framework[1] in Python. The inference application is deployed on an Azure Kubernetes Service (AKS)[2] cluster (see Figure 1). When a user inputs a question, the inference application processes it. The application first employs a query_rewording function to replace acronyms with their full forms, avoiding ambiguities in the query (e.g., D.C. to Death Certificate). The expanded query is then formatted for Azure AI vector search to retrieve the relevant top K chunks from both unstructured and structured indexes, which are combined in the prompt as context for further use (see Figure 1). Here K is subjective, we chose K values based on priority of our data sources in the range of 2-4. An example prompt is shown below.

System: You are a call center agent answering customer questions. Answer the following question based on the information provided in the following CONTEXT.
-If the CONTEXT is EMPTY, please state "I cannot answer this question based on the available information"
-If the CONTEXT in NOT EMPTY, MAKE SURE to consider all the sources to answer the question. Indicate in parentheses the source numbers for each answer bullet point.
-For answers with a single word or number, answer within a brief sentence.
#CONTEXT { "Source":1, "Policy Number": "******", "Section Name":"***", "Product Rules": "...covered by policy rules...", ... }
User: {{#QUESTION: What are the product rules for this policy?}}
{{#Output_format: Answer in bullet points}}

We engineered our prompt with the *RACE* framework (Liu et al., 2023) to ensure accurate answer generation, adding instructions to prevent hallucinations, expanding one-word answers into full sentences, and identifying the source of each answer from the context. Users can choose output formats such as paragraphs or bullet points, with sources listed at the end of bullet points to trace answers and mitigate hallucinations. We used Azure OpenAI's GPT-3.5-turbo model[3] for this process.

To avoid hallucinations in generated answers, in addition to the guardrails and source references, we also validate answers with a secondary LLM (GPT-4 in our application). A special prompt rates the groundedness of answers on a scale of 1 to 5, employing few-shot prompting techniques with examples of both good, partially good, and bad answers provided in JSON format. This final validation process reduces hallucinations and informs users about confidence ratings (groundedness) and rationales. An illustrative validation prompt is shown below.

System: You are an answer validation assistant. You will be given a CONTEXT and an ANSWER. The CONTEXT is composed of various source pieces .....
User: Your evaluation should be based on the following rating scale: ......
Independent Examples:
Example 1 Input: {"CONTEXT": '{"policy number": "***", "Type": "Regular", "lifetime value": *******}', "ANSWER": "Your benefit type is "SuperCare".}
Example 1 Output: {answer: 1, reason: "The answer contains information not present in the context."} ......

3.3 Monitoring and LLM Operations

To ensure efficient operation of our application, we automated its deployment and incorporated comprehensive monitoring functionalities, including application logging, data monitoring, continuous integration and deployment (CI/CD), and model monitoring (see Figure 1). Application logs are sent to a Devo server to aid in debugging issues such as crashes or latency. Data monitoring involves versioning data sources upon ingestion and assessing their quality using checks for null values, data types, and parsability. We also version prompts to maintain consistency and reliability as the prompts(or LLM) evolve. For CI/CD, TeamCity [4]is used to automatically deploy the application on an AKS cluster upon code changes in Git repositories.

Model monitoring includes content logging on the user interface, where we capture CSRs' questions, generated answers, and confidence ratings from the secondary LLM. This is supplemented with optional CSRs' feedback on answer accuracy and completeness. A statistics dashboard in PowerBI analyzes this data, identifying trends and quality issues in generated answers. This helps maintain high customer satisfaction by addressing

[1]https://microsoft.github.io/promptflow/
[2]https://azure.microsoft.com/en-us/products/kubernetes-service
[3]https://learn.microsoft.com/en-us/azure/ai-services/openai/concepts/models
[4]https://www.jetbrains.com/teamcity/

low-feedback and low-confidence answers.

3.4 User Interface

The user interface for CSRs is easy-to-use, featuring a text box for questions and a list of frequently asked questions (see Figure 4 in Appendix). CSRs can type questions or select from the list and must provide a policy number or related information to receive customer-specific answers. The interface displays answers with source lists, summaries, and clickable URLs for quick navigation. The source numbers are cited in the answer for easy validation as shown in Figure 5 in the appendix. We present both structured and unstructured data sources and users can submit feedback on answer quality.

4 Evaluation and Discussion

4.1 Evaluation

In this study, to demonstrate the effectiveness of our proposed methodology, we performed evaluations by surveying users' feedback on answers' accuracy and completeness. The business users are either CSRs or their managers who are familiar with the products and are considered as subject matter experts. We implemented a feedback mechanism where users could rate each answer's **accuracy** and **completeness** on a scale of 1-5 by clicking one of the five stars on the user interface. Meanwhile, our validation model depicted in Section 3 rates **confidence** on the same scale. We also evaluate the **response latency** of our inference pipeline to highlight the rapid response time of our application. To perform this evaluation, we extract users' activity data for 26 weeks from May 13 to Nov 10, 2024 with a total of 27471 queries, among which 1302 received feedback. We plotted the **weekly averaged** metrics [1] in Figure 2.

User Feedback Evaluation: Figure 2 shows the weekly average feedback from users on accuracy and completeness. It also shows the weekly average response times and the weekly average confidence ratings of the secondary LLM. It can be observed that weekly averages for accuracy and completeness remained high (3 to 4 star ratings) in most of the weeks, except for weeks 12 to 14. The confidence ratings of the secondary LLM remained greater than 4 in all weeks.

[1]We exclude the cases with null response times from all analyses. Additionally, for the response time analyses, any outliers falling outside the Inter-Quartile Range are also removed. Due to the limited space in the paper, we plot **weekly averaged** metric, instead of individual log record in this figure.

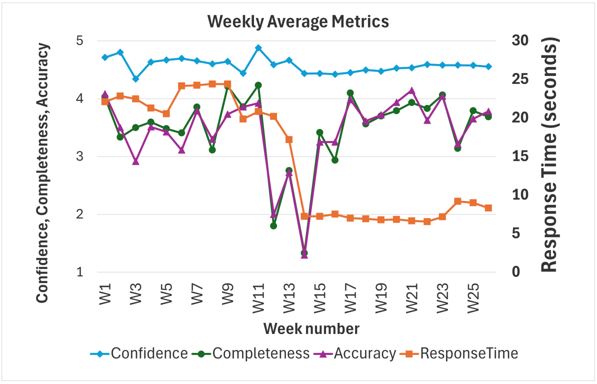


Figure 2: Weekly Average of Metrics over Week 1-26

In Figure 2, we observe that the accuracy and completeness rating dropped between week 12 and week 14. This occurred when CSRs were searching for answers on a policy that was not present in the index, and the prompt also needed an adjustment to avoid the generation of an ambiguous answer from another source. Once the missing data was ingested, the accuracy and completeness feedback improved again.

We further analyzed the data and observed that 52.07% of the responses received 5-star ratings for accuracy and 53.69% of the answers received 5-star ratings for completeness. The confidence ratings are 5 stars 77.83% of the times; showing that majority of the times secondary LLM was having the same opinion as the primary LLM.

Higher scores on the metrics throughout the production evaluation period demonstrates that the answers are consistently reliable and that business users could adopt them confidently. Our application reduces CSRs' workload and minimizes the risk of overlooking information, a significant improvement over the previous system, where CSRs were required to sift through multiple knowledge bases on different screens and read policy documents.

Response Time: We also monitored the weekly average response time during the same evaluation period (measured in seconds) as shown in Figure 2. We can observe that during week 1 to week 13, the average response time hovers around 20 seconds with an average of 21.91s. To reduce response time, on Week 14, we improved the retrieval step from the database index by discarding vectors (embeddings) from the retrieved results and only retrieved text of the relevant chunks with metadata for prompt generation. This optimization significantly improved the response time. Note that we already implemented multi-threading for data retrieval and switched to higher tier subscription of Azure AI Search (vector database). After week 14, our weekly average response time ranges between 6.56s and 9.19s, with an average of 7.33s. This is a significant improvement in response time.

To further illustrate, our application achieves such low latency in generating a response during inference time, we pick 35 random execution records between August 2024 and November 2024 from our execution log and calculate the average the execution time for each step. The averaged step-wise latency is presented in Figure 3.

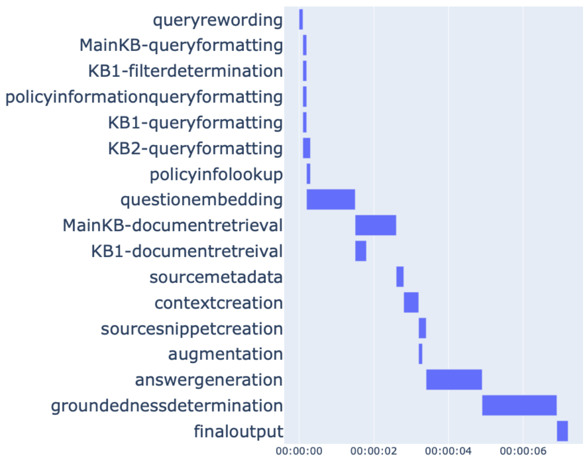


Figure 3: Latency Decomposition by Steps

In Figure 3, we can observe that user queries received a valid response within 7.20s. This is an impressive response time considering the number of steps in the entire RAG pipeline. The first few steps of query pre-processing take a few milliseconds, then question embedding and document retrieval take 1.3s and 1.1s respectively. The retrieved snippets of context are then passed to the answer generation step, which takes 1.5s, and the final groundedness (confidence) validation step takes 2s in the execution of the whole pipeline. It is to be noted that we have used higher tier of Azure AI Search (tier L2, 12 partitions, 24 search units) and Azure Open AI PTU (Provisioned Throughput Unit)[1] and optimized the retrieval step by retaining only the metadata and text chunk for improved performance in a production application.

Comparison of LLMs on Answer Generation: We also compared multiple LLMs for answer generation. Our method of comparison is as follows. We first labeled the ground truth answers by collecting user feedback on the answer generated by GPT 3.5 model. We picked those answers where the user generated a feedback rating of 5 star on both accuracy and completeness. These are about 35 questions and their answers. We then generated the same answer using other popular open source LLMs and GPT family's LLMs. Our list of LLMs include: GPT-4, GPT-4o, LLama-3, Mistal-large, Mistral-small, Micorosoft's Phi128-small. Although this is not a comprehensive list, it provides a good understanding of industrial study. We also tried some other models not in this list but they hallucinated in preliminary tests so we excluded them from our comparison, such as Dolly-v2, and Cohere's LLM. We used three metrics to compare them: ROUGE (Briman and Yildiz, 2024), BLEU (Reiter, 2018) and cosine similarity (Dehak et al., 2010) scores. These three metrics are popular metrics in the literature for comparing generative text against a bench mark. Our results are shown in Table 2. It can be observed from Table 2, LLama-3 and GPT-4o are closest to answer generation compared to GPT 3.5. Mistral-small also shows some impressive performance despite its smaller model size (22B). Those results demonstrate that quality of answer generation is not dependent on the model size but on the type of data it was trained/fine-tuned on. This comparison also helped us to decide which models can be used to replace the other ones for answer generation and helped us control the cost.

Table 2: LLM Comparison Results

Model	Avg Bleu Score	Avg Rouge Score	Avg Cossim Score
Llama-3-70B	0.279	0.421	0.838
Mistral-Large	0.101	0.333	0.698
Phi128-Small	0.193	0.283	0.700
Mistral-Small	0.207	0.376	0.785
GPT-4o	0.397	0.492	0.784

4.2 Discussion and Limitations

Although our framework achieves high accuracy, low latency, and strong groundedness in question-answering on a large structured dataset, it does have its limitations. One limitation is the time required for the offline indexing step compared with the text-to-SQL method with an LLM function call. This text-to-SQL method can directly leverage existing structured data stored in the databases at inference time without indexing. Our method requires an offline embedding and indexing step to convert

[1]https://learn.microsoft.com/en-us/azure/ai-services/openai/how-to/provisioned-throughput-onboarding

the structured data into a searchable vector index. This step may take longer if the size of the data is large. This is a trade-off between the higher accuracy and lower response time at inference time versus the delay at the data ingestion stage. We mitigated this impact by aggregating our structured data to reduce the number of calls for embedding model. We also improved our data indexing method by using parallelization in the code. In addition, when structured data changed, we identified the change using keys and only updated vectors for the changes. In case of aggregation level questions (count, sum, group by) for this approach, it is better to list them in advance and index data in a way that it can be answered faster at inference time.

5 Conclusion and Future Work

In this paper, we presented an industrial case study on the implementation of RAG technique. We presented a novel enhancement to the RAG technique by transforming structured data to JSON format and then embedding it in the same way as unstructured data for faster and accurate answer generation. We also showed a comparison of popular open-source and closed-source LLMs on answer generation in our business case. We conclude that lower response time and highly accurate answers can be retrieved using our approach combined with scalable infrastructure. We also conclude that LLMOps is important for industrial applications and helps in maintaining the high quality of answer generation. We also conclude that LLama-3, GPT-4o and Mistral small are as good as GPT-3.5 in answer generation.

Our proposed methodology is highly generalizable and could be easily applied to other business use cases, where both structured and unstructured data are queried to generate a grounded answer. In the future, we will expand the application to serve other business lines such as presale consulting services, where sales agents need access to both unstructured knowledge articles and product specifications stored in structured databases. In addition to serving financial institutions, our application can be readily adapted for other industries, such as healthcare institutions where a large amount of structured and unstructured medical data needs to be leveraged to answer a complex question. We hope that this work can provide insights into the use of both structured and unstructured data in an end-to-end manner in RAG applications and inspire new advanced RAG applications in industry.

References

Karam Ahkouk, Mustapha Machkour, Khadija Majhadi, and Rachid Mama. 2021. A review of the text to SQL frameworks. In *NISS2021: The 4th International Conference on Networking, Information Systems & Security, KENITRA, Morocco, April 1 - 2, 2021*, pages 45:1–45:6. ACM.

Cagri Balkesen, Nitin Kunal, Georgios Giannikis, Pit Fender, Seema Sundara, Felix Schmidt, Jarod Wen, Sandeep R. Agrawal, Arun Raghavan, Venkatanathan Varadarajan, Anand Viswanathan, Balakrishnan Chandrasekaran, Sam Idicula, Nipun Agarwal, and Eric Sedlar. 2018. RAPID: in-memory analytical query processing engine with extreme performance per watt. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 1407–1419. ACM.

Mohammed Khalid Hilmi Briman and Beytullah Yildiz. 2024. Beyond ROUGE: A comprehensive evaluation metric for abstractive summarization leveraging similarity, entailment, and acceptability. *Int. J. Artif. Intell. Tools*, 33(5):2450017:1–2450017:23.

Jiawei Chen, Hongyu Lin, Xianpei Han, and Le Sun. 2024. Benchmarking large language models in retrieval-augmented generation. In *Thirty-Eighth AAAI Conference on Artificial Intelligence, AAAI 2024, Thirty-Sixth Conference on Innovative Applications of Artificial Intelligence, IAAI 2024, Fourteenth Symposium on Educational Advances in Artificial Intelligence, EAAI 2014, February 20-27, 2024, Vancouver, Canada*, pages 17754–17762. AAAI Press.

Najim Dehak, Réda Dehak, James R. Glass, Douglas A. Reynolds, and Patrick Kenny. 2010. Cosine similarity scoring without score normalization techniques. In *Odyssey 2010: The Speaker and Language Recognition Workshop, Brno, Czech Republic, June 28 - July 1, 2010*, page 15. ISCA.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, pages 4171–4186. Association for Computational Linguistics.

Tom B. Brown et al. 2020. Language models are few-shot learners. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual.*

Stephane Faroult and Peter Robson. 2006. *The art of SQL.* O'Reilly.

Paulo Finardi, Leonardo Avila, Rodrigo Castaldoni, Pedro Gengo, Celio Larcher, Marcos Piau, Pablo B. Costa, and Vinicius Fernandes Caridá. 2024. The chronicles of RAG: the retriever, the chunk and the generator. *CoRR*, abs/2401.07883.

Jonathan Herzig, Pawel Krzysztof Nowak, Thomas Müller, Francesco Piccinno, and Julian Martin Eisenschlos. 2020. Tapas: Weakly supervised table parsing via pre-training. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020*, pages 4320–4333. Association for Computational Linguistics.

Wonseok Hwang, Jinyeung Yim, Seunghyun Park, and Minjoon Seo. 2019. A comprehensive exploration on wikisql with table-aware word contextualization. *CoRR*, abs/1902.01069.

Vladimir Karpukhin, Barlas Oguz, Sewon Min, Patrick S. H. Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. 2020. Dense passage retrieval for open-domain question answering. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing, EMNLP 2020, Online, November 16-20, 2020*, pages 6769–6781. Association for Computational Linguistics.

LangChain. 2024a. Build a question answering application over a graph database.

LangChain. 2024b. Build a question/answering system over sql data.

LangChain. 2024c. Tool calling.

Oleg Lashinin, Kirill Bykov, Marina Ananyeva, and Sergey Kolesnikov. 2023. Gpt3recbot: a universal chatbot recommender of movies, books and music in telegram. In *Proceedings of the Fifth Knowledge-aware and Conversational Recommender Systems Workshop co-located with 17th ACM Conference on Recommender Systems (RecSys 2023), Singapore, September 19th, 2023*, volume 3560 of *CEUR Workshop Proceedings*, pages 35–43. CEUR-WS.org.

Wenqiang Lei, Weixin Wang, Zhixin Ma, Tian Gan, Wei Lu, Min-Yen Kan, and Tat-Seng Chua. 2020. Re-examining the role of schema linking in text-to-sql. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing, EMNLP 2020, Online, November 16-20, 2020*, pages 6943–6954. Association for Computational Linguistics.

Xiaoxia Liu, Jingyi Wang, Jun Sun, Xiaohan Yuan, Guoliang Dong, Peng Di, Wenhai Wang, and Dongxia Wang. 2023. Prompting frameworks for large language models: A survey. *CoRR*, abs/2311.12785.

Solmaz Seyed Monir, Irene Lau, Shubing Yang, and Dongfang Zhao. 2024. Vectorsearch: Enhancing document retrieval with semantic embeddings and optimized search. *CoRR*, abs/2409.17383.

Bowen Qin, Binyuan Hui, Lihan Wang, Min Yang, Jinyang Li, Binhua Li, Ruiying Geng, Rongyu Cao, Jian Sun, Luo Si, Fei Huang, and Yongbin Li. 2022. A survey on text-to-sql parsing: Concepts, methods, and future directions. *CoRR*, abs/2208.13629.

Renyi Qu, Ruixuan Tu, and Forrest Sheng Bao. 2024. Is semantic chunking worth the computational cost? *CoRR*, abs/2410.13070.

Ehud Reiter. 2018. A structured review of the validity of BLEU. *Comput. Linguistics*, 44(3).

Mohammad Shahin, F. Frank Chen, and Ali Hosseinzadeh. 2024. Harnessing customized AI to create voice of customer via GPT3.5. *Adv. Eng. Informatics*, 61:102462.

Kevin Stower and Dirk Krechel. 2019. Seq2sql - evaluating different deep learning architectures using word embeddings. In *Machine Learning and Data Mining in Pattern Recognition, 15th International Conference on Machine Learning and Data Mining, MLDM 2019, New York, NY, USA, July 20-25, 2019, Proceedings, Volume I*, pages 343–354. ibai Publishing.

Tao Yu, Zifan Li, Zilin Zhang, Rui Zhang, and Dragomir R. Radev. 2018. Typesql: Knowledge-based type-aware neural text-to-sql generation. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT, New Orleans, Louisiana, USA, June 1-6, 2018, Volume 2 (Short Papers)*, pages 588–594. Association for Computational Linguistics.

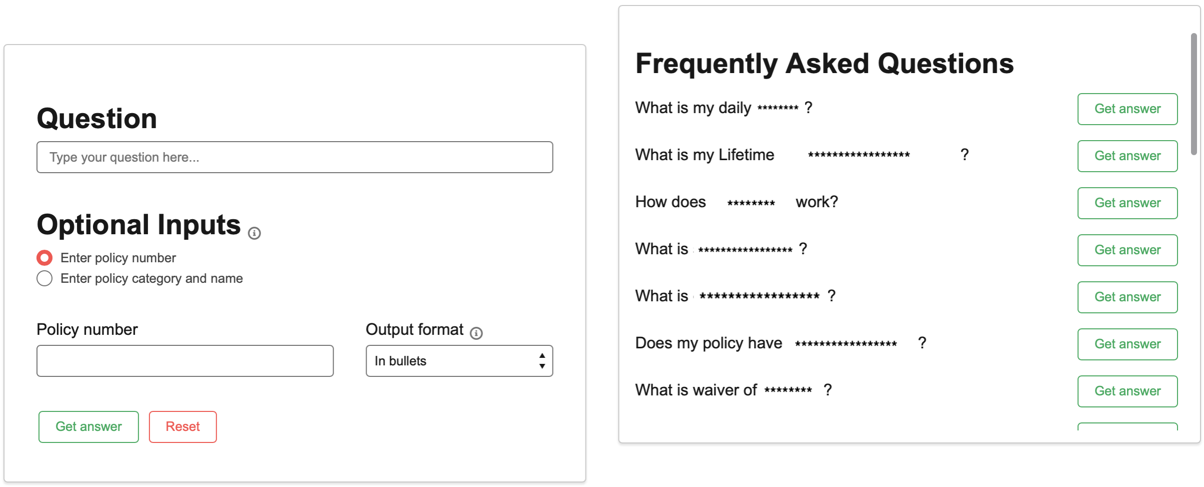
Zihan Zhang, Meng Fang, and Ling Chen. 2024. Retrievalqa: Assessing adaptive retrieval-augmented generation for short-form open-domain question answering. In *Findings of the Association for Computational Linguistics, ACL 2024, Bangkok, Thailand and virtual meeting, August 11-16, 2024*, pages 6963–6975. Association for Computational Linguistics.

Zijie Zhong, Hanwen Liu, Xiaoya Cui, Xiaofan Zhang, and Zengchang Qin. 2024. Mix-of-granularity: Optimize the chunking granularity for retrieval-augmented generation. *CoRR*, abs/2406.00456.

Kun Zhu, Xiaocheng Feng, Xiyuan Du, Yuxuan Gu, Weijiang Yu, Haotian Wang, Qianglong Chen, Zheng Chu, Jingchang Chen, and Bing Qin. 2024. An information bottleneck perspective for effective noise filtering on retrieval-augmented generation. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024*, pages 1044–1069. Association for Computational Linguistics.

A Appendix

The user interface of the application is shown in Figure 4 and Figure 5.



Question

Type your question here...

Optional Inputs

[✓ Enter policy number]
Enter policy category and name

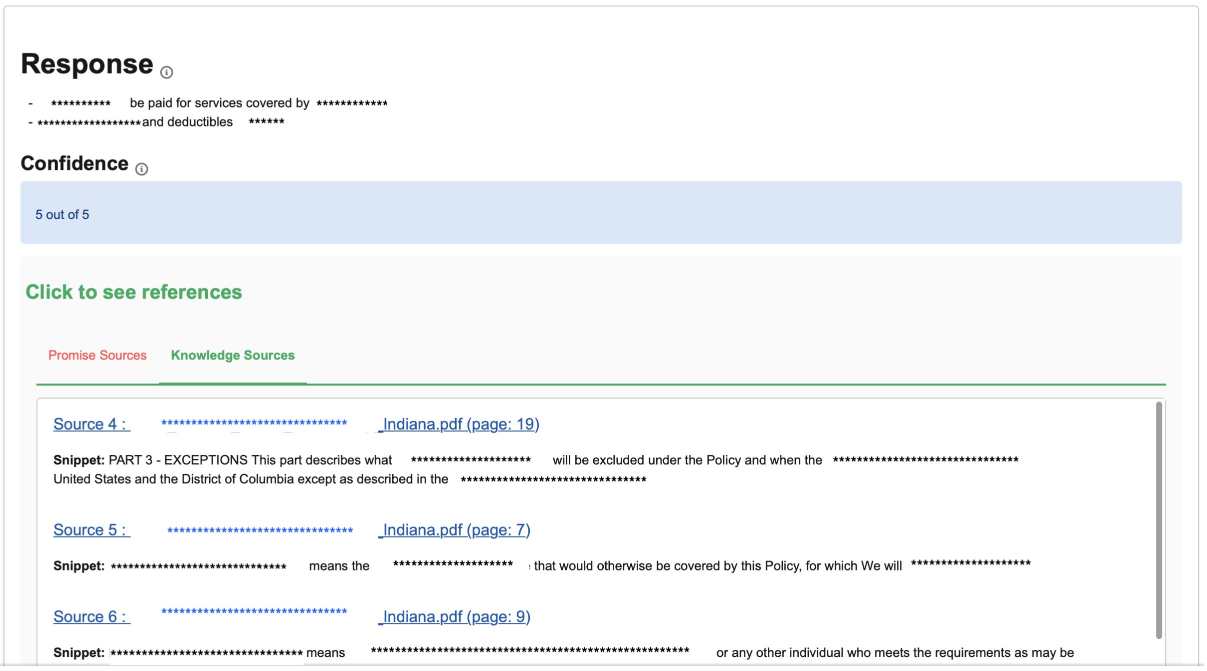
Policy number

Output format: In bullets

Frequently Asked Questions

What is my daily ******** ?
What is my Lifetime ***************** ?
How does ******** work?
What is **************** ?
What is ***************** ?
Does my policy have ***************** ?
What is waiver of ******** ?

Figure 4: GUI of the Application: Input Section



Response

- ********** be paid for services covered by ************
- ******************and deductibles ******

Confidence

5 out of 5

Click to see references

Source 4 : ******************************* Indiana.pdf (page: 19)

Snippet: PART 3 - EXCEPTIONS This part describes what ******************** will be excluded under the Policy and when the ******************************* United States and the District of Columbia except as described in the *******************************

Source 5 : ******************************* Indiana.pdf (page: 7)

Snippet: ****************************** means the ******************** that would otherwise be covered by this Policy, for which We will ********************

Source 6 : ******************************* Indiana.pdf (page: 9)

Snippet: ******************************** means ** or any other individual who meets the requirements as may be

Figure 5: GUI of the Application: Answer and References