Enter: ADIPEC 2024

Enter: SPE-222032

Enhancing Enterprise-Wide Information Retrieval through RAG Systems Techniques, Evaluation, and Scalable Deployment

Somasundaram Palaniappan[1] , Rohit Mali [1] , Luca Cuomo[2] , Mauro Vitale[2] , Ali Youssef[2] , Athira Puthanveetil Madathil[1] , Malathi Murugesan[1] , Alessandro Bettini[2] , Giovanni De Magistris[2] , Giacomo Veneri[2]

[1]Baker Hughes, Bangalore, India
[2]Baker Hughes, Florence, Italy

Abstract

Organizations in the Oil and Gas Industry deal with a vast repository of knowledge dispersed across various structured and unstructured data sources. The process of retrieving information for a particular task can be time-consuming and necessitates the involvement of multiple subject matter experts. Large Language Models (LLMs) supported by Retrieval Augmented Generation (RAG) systems can help expedite information retrieval and processing. However, Enterprise-scale LLM implementation faces challenges, that include confidentiality, scalability, handling hallucinations, handling domain data, feedback, and model improvement.

In this technical paper, we describe an optimized RAG system using LLMs for accelerated information retrieval from vast, highly confidential sources. We present the software architecture, the data flow that facilitates the application's ability to gather feedback for model fine-tuning and the techniques used to optimize our solution for accuracy, scalability, throughput. In the proposed architecture, the LLM-as-a-service component generated an average throughput of 989 tokens per second for summarization tasks involving up to 2500 tokens per request when run on A100 GPUs (40 GB). In the Oil & Gas Industry, the application allows Field Service Engineers to directly retrieve technical details about machinery, details from maintenance manuals etc and helps design teams to speed up information retrieval. This has resulted in an estimated improved productivity in the range of 20% and a positive impact on Customer Relationship, Time to Serve and Cost of Quality.

1. Introduction

Ensuring timely availability of data helps businesses streamline their operations and improve operational efficiency and leads to improved customer satisfaction. For instance, in the Oil and Gas Industry, when Field Service Engineers require technical details about operating conditions or some specific information from maintenance manuals, they often need to collaborate with the Service Engineering team having the subject matter experts, which may be in a different time zone. This often results in a lot of back-and-forth communication leading to extended wait times for customers and increased Cost of Quality. In the area of Specification Management, in the case of new projects, application and design engineers spend a considerable amount of time in screening technical specifications, standards and design practices to align with customer requirements. Streamlining information flow has a potential to reduce machinery downtime and will help on time delivery for maintenance.

LLMs demonstrate a high level of fluency and comprehension in question answering, text summarization, open-ended conversations and even code generation. However, despite their achievements, LLMs have their limitations when dealing with domain-specific or knowledge-intensive tasks [1,2]. LLMS hallucinate [1,3] when handling queries beyond their training data or requiring current information. One effective response to these issues is the use of Retrieval-Augmented Generation (RAG). RAG improves the capabilities of Large Language Models (LLMs) by retrieving pertinent information from external knowledge bases. This data is then integrated into the LLM through engineered prompts, which help to anchor the LLM in reality and significantly decrease the occurrence of hallucinations. An application built with RAG+LLM has high level of comprehension, fluency and is also grounded and this has led to the widespread adoption of RAG+LLM systems.

We have commercially available RAG+LLM systems but it is preferred to process highly confidential and export-controlled data locally, preventing unauthorized access to data. This is a very important feature for any industrial Company and specifically in the O&G sector where operational data representing the Company value and competitive advantage must be secured and protected as a vital asset that cannot be handed over to third party players.

In this technical paper we present a RAG+LLM system with a chatbot like UI that can be locally hosted, connected to company knowledge base and serve information on-demand. We discuss in detail the following:

- Components of our RAG
- The software architecture of our RAG system and deployment strategies for ensuring scalability and reliability.
- Microservices approach to make LLMs and RAG reusable for various cases; LLM-as-a-service, RAG-as-a-service, Embedding-as-a-service, ReRank-as-a-service.
- Feedback loop to allow LLM finetuning.
- Benchmarking metrics to assess the performance of our system.

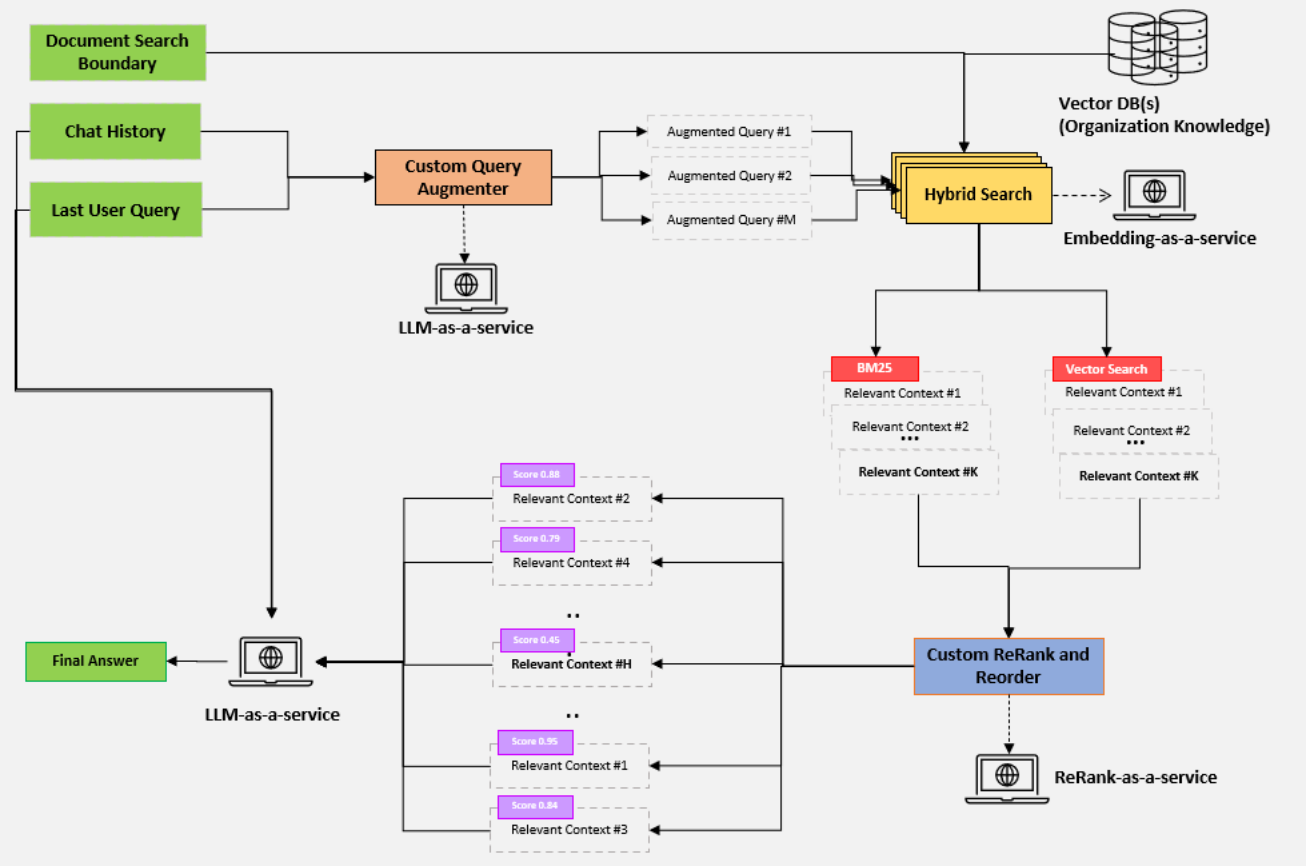

Fig. 1 RAG System Overview. First Custom Query Augmenter creates different variations of the user query that is also history aware (brown box). Then the set of augmented queries are then used along with document search boundary to fetch relevant contexts from knowledge base (yellow box). The queried contexts are first re-ranked and then re-ordered (blue box). Finally, the re-ordered contexts along with original inputs are provided to LLM to generate the response.

2. Retrieval Augmented Generation (RAG) Components

This section describes the sequence of processes involved in our RAG system, which is depicted in Fig. 1.
The processes can be outlined as follows:

1. The process starts with the user query and the chat history of the current session serving as inputs to the Custom Query Augmenter.
2. The **Custom Query Augmenter** takes these inputs and generates multiple variations of the query that is also aware of any references to the user session history.
3. The **Hybrid Search** component retrieves relevant contexts from the knowledge base using the set of augmented queries and the search boundary input.
4. The **Custom ReRank and Reorder** component identifies and presents search results in a specific order, ensuring they are highly relevant to the user's query and effectively addressing the "Lost in the Middle" issue.
5. Finally, the user query, chat history, and the rearranged contexts are fed as inputs to the Language Model (LLM) to generate the results.

Fig 1 illustrates the modular design of the RAG system, which enables us to swap out or incorporate more sophisticated methods in particular stages without disrupting the others. For example, for workflows where chat history is not required, the Custom Query Augmenter can be replaced with another component that does not look at chat history. The RAG system relies on several external services, including LLM-as-a-service, Embedding-as-a-service, and ReRank-as-a-service. By delivering common components as Software-As-A-Service, we can optimize resource usage, enhance GPU efficiency, and increase reusability. The default models for LLM-as-a-service, Embedding-as-a-service, and ReRank-as-a-service are mistral-7b-instruct-v0.2, gte-large-en-1.5, and bge-reranker-large, respectively. These services will be discussed in greater detail in subsequent sections.

2.1. Custom Query Augmenter

During a multi-turn exchange between a user and a system, a user's query may be succinct and ambiguous due to direct or indirect references to previous interaction. Under such circumstances, the system's context retrieval performance may not meet expectations. Query expansion has been used in search for years [4]. By incorporating essential information from past interactions into the search query, the search results can be significantly improved. To further enhance search performance, generating multiple query rewrites for the same query can be an effective approach, as demonstrated in previous research [6].

The bespoke Custom Query Augmenter employs a generative approach using LLM-as-a-service to generate multiple query variations of the user query. The generated revisions incorporate essential information from user chat history. The list of generated queries are passed as inputs to the next stage Custom ReRank and Reorder.

2.3. Hybrid Search

Due to the context length limitation of LLM (Large Language Model), it is more efficient to provide only the relevant sources for processing, rather than large data chunks. Consequently, a hybrid search approach is often preferred over relying solely on LLM long context processing capabilities.

Hybrid search refers to a search methodology that integrates two or more distinct search algorithms to enhance the accuracy and relevance of retrieved information. Hybrid search typically integrates the conventional keyword search method with advanced vector search. Our Hybrid Search Component combines BM25 [8] keyword search and dense vector similarity search to produce the search results. The set of augmented queries along with the defined search boundary is used to retrieve the most relevant contexts from a restricted search area. By limiting the search area, we ensure that users can only access the data that is relevant to their role or authorization level. This restriction helps maintain data security and privacy by preventing unauthorized access to sensitive information. The search results are passed as inputs to the next stage Custom ReRank and Reorder.

2.4. Custom ReRank and Reorder

The study titled 'Lost in the Middle' reveals that the performance of Language Models (LLMs) can deteriorate substantially when shifting the location of crucial information within lengthy contexts. This finding underscores the fact that existing language models are not reliably harnessing information from extended input sequences [7]. Furthermore, the outcomes of Hybrid Search encompass both relevant contexts and unwanted elements that will result in sub optimal outputs from LLM.

To address the issues highlighted above the bespoke Custom Rerank and Reorder component employs a transformer-based model combined with other techniques to select and return the top-n reordered results to reduce the presence of irrelevant information and to mitigate 'Lost in the Middle' problem.

2.5. Generating the final response

The reorganized set of relevant company knowledge produced by Custom Rerank and Reorder module along with the original user query and the chat history is fed to LLM-as-a-service using a prompt to generate the user response. The prompt includes prompt level guardrails and is carefully engineered to answer only from the provided contexts.

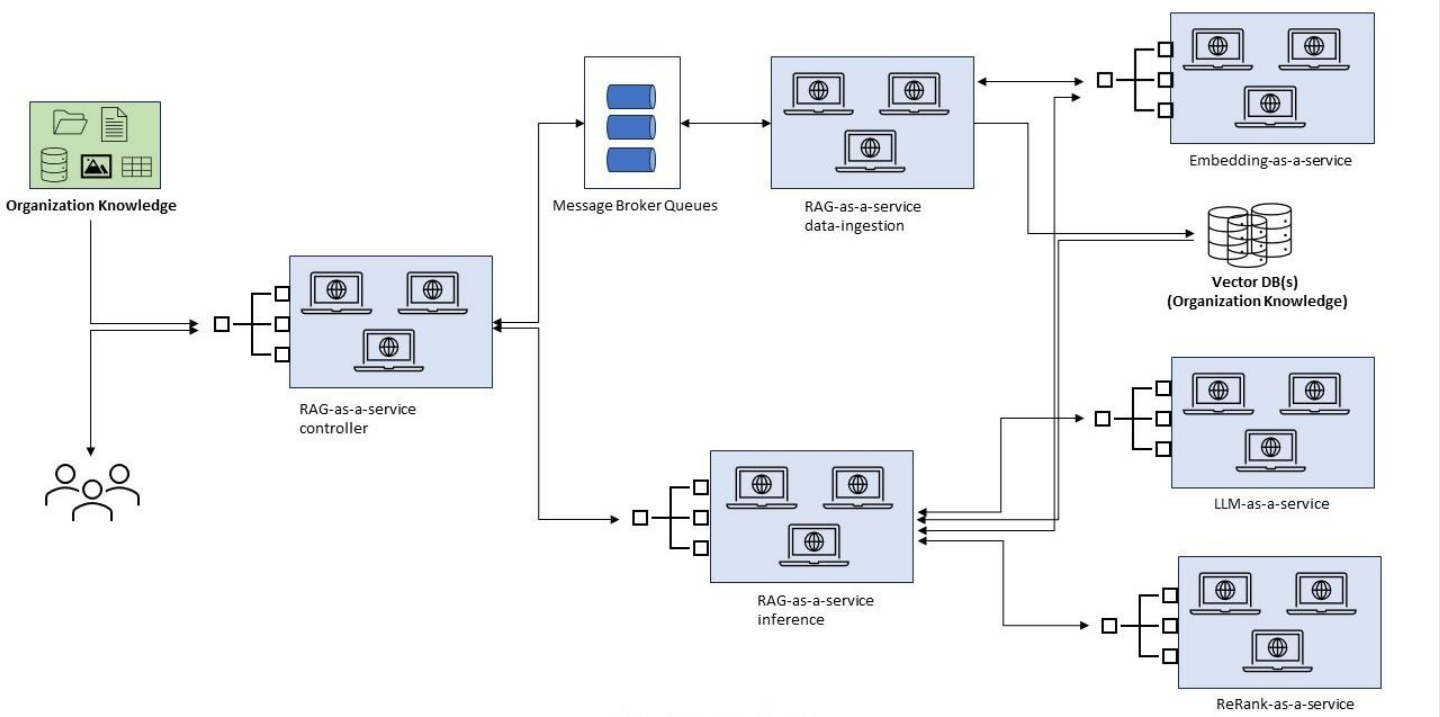

Fig. 2 RAG-as-a-service Architecture. This is a load balanced, scalable microservices architecture (blue boxes). RAG-as-a-service controller orchestrates the incoming requests and forwards the requests to RAG-as-a-service data-ingestion or RAG-as-a-service inference. RAG-as-a-service data-ingestion converts raw data into chunks of dense vector representations and indices. RAG-as-a-service inference helps answer user queries, the architecture is in Fig. 1

3. Software Architecture and Deployment Strategies

The integration of RAG with LLM enables the extraction of crucial information from external knowledge bases. This functionality is versatile and applicable to various business applications within the organization. For instance, a FieldService engineer might want to upload a maintenance manual or some other document and ask queries about it. Another scenario could involve machine-related data located in a specific area, where the Field Service Engineer may wish to retrieve machine data without uploading documents in real-time. RAG-as-a-Service is a horizontal solution that facilitates the extraction of valuable information from controlled search areas. It can be seamlessly integrated into custom vertical applications through API where the search area can be dynamically configured based on business requirements.

An enterprise scale RAG application requires a robust architecture to handle large volumes of data, significant loads, and the need for flexibility to accommodate customized workflows. These demands led us to opt for a microservices design over other architectural patterns, such as a monolith or a modular monolith. This design pattern also attracts its own set of challenges which will be discussed in upcoming sections.

Fig. 2. illustrates the architecture for RAG-as-a-service, which is built using a microservices design. The following services are included in this setup:

1. RAG-as-a-service controller
2. RAG-as-a-service Inference
3. RAG-as-a-service data-ingestion
4. LLM-as-a-service
5. ReRank-as-a-service
6. Embedding-as-a-service

The upcoming sections will give a more detailed explanation of the functions each microservice is designed to handle.

3.1. RAG-as-a-service controller

The RAG-as-a-service controller acts as the orchestrator of the RAG system, handling and directing incoming requests. It determines whether to forward these requests to the RAG-as-a-service Inference or RAG-as-a-service data-ingestion components. The controller plays a crucial role in distributing data ingestion tasks evenly among all available runners and monitoring the status of all ongoing jobs. Additionally, it oversees the management of vertical applications and their allocated storage space.

3.2. RAG-as-a-service Inference

The RAG-as-a-service inference functions as a query answering tool within the permissive search domain. It processes user queries and search boundaries to deliver the desired results. The underlying structure and data flow are depicted in Fig. 1. The detailed description can be found in Section 2.

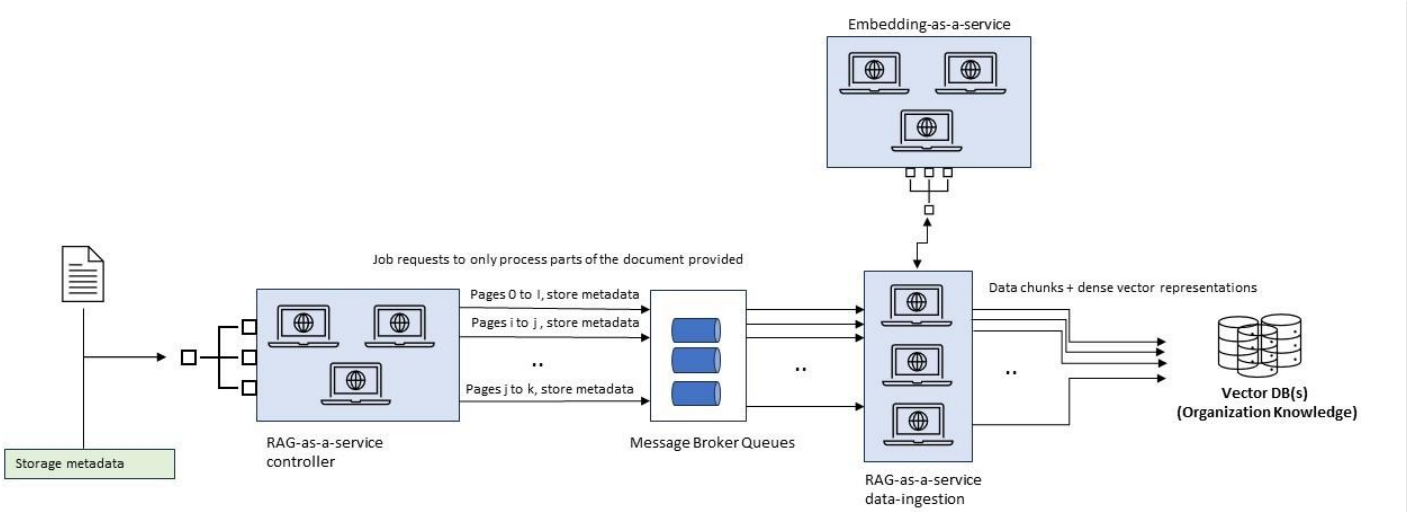

Fig. 3 RAG-as-a-service data-ingestion workflow. RAG controller splits a data-ingestion job into n jobs, which is then processed by the pool of data-ingestion workers and converted to dense vector representations and search indices.

3.3. RAG-as-a-service data-ingestion

The RAG system is unable to utilize company data in its original state. Instead, it relies on search indices and dense vector presentations of data to function effectively through its hybrid search component. The data-ingestion service plays a crucial role in this process by transforming raw data into searchable formats.

To ensure efficient and resilient data ingestion, an asynchronous queue-based architecture is employed. The workflow for data ingestion is depicted in Fig 3. In this setup, a solitary data file is fragmented into multiple data ingestion tasks by the RAG Controller. These tasks are subsequently dispatched to the Job Queues within the message broker, which are grabbed by the data ingestion workers. This design enables parallel processing of the same data across distinct nodes, thereby enhancing the speed of data ingestion. Moreover, the queue-based architecture provides robustness. In the event of a data ingestion worker failure or crash, the job designated to that worker is automatically retried on other functional workers, up to a maximum of k retries. This mechanism ensures data integrity and prevents data loss.

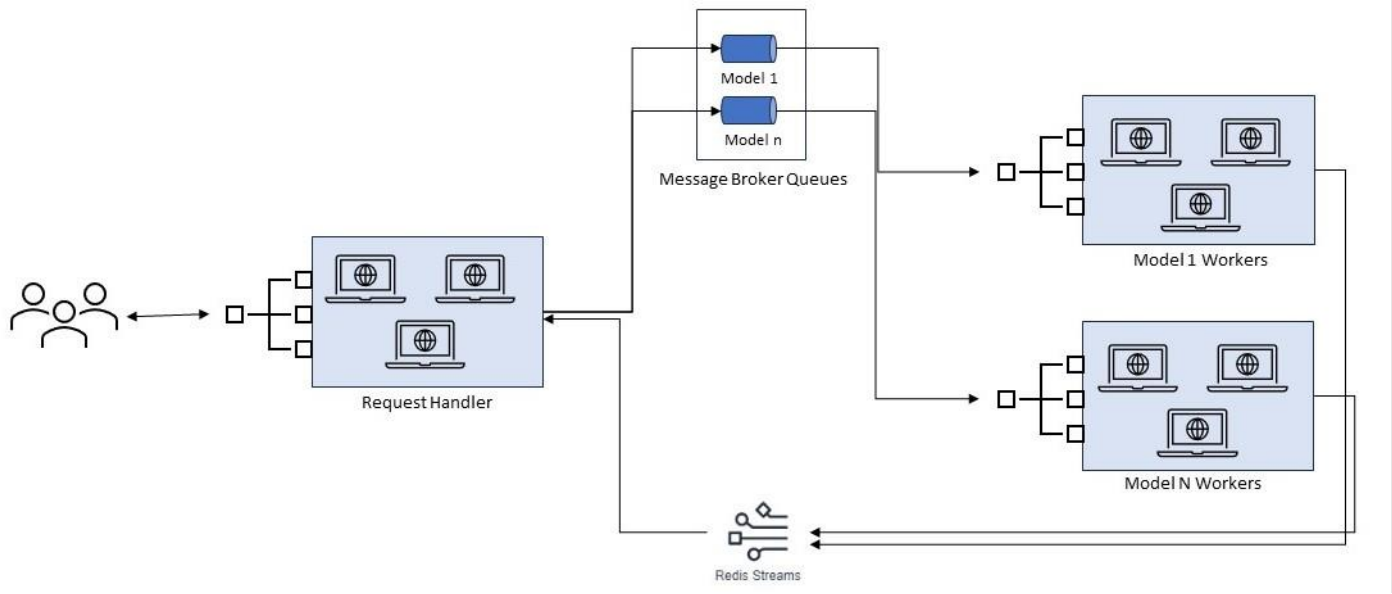

Fig. 4. LLM-as-a-service Architecture. Request Handler despatches requests to the correct model and streams response to the client.

3.4 LLM-as-a-service

In the real world, the expense of running most Generative AI applications primarily stems from the compute requirements of Large Language Models (LLMs). If we embed LLMs into the application that necessitates their use, we risk underutilizing and wasting expensive GPU resources. We will also increase both the coding and deployment intricacy of the application due to the embedded LLM.

By hosting LLMs as a service, we can not only optimize their usage for minimal latency, maximum throughput, and resource conservation but also enable the utilization of LLMs for functions beyond just RAG. LLM-as-a-service adopts OpenAI API specifications to be compatible with most opensource frameworks.

Fig. 4 illustrates the structure of LLM-as-a-service. In our microservices architecture, there are two primary components: a Request Handler and one or more Model Workers. Overall, the LLM Request Handler distributes requests to designated model queues. Model workers, in turn, retrieve tasks from these queues and transmit responses in a continuous flow using Redis Streams. This queue-based design offers several advantages, including the ability to independently scale multiple LLM models and enhanced fault tolerance through request retries. By managing scaling based on acceptable latency, queue backlog, and the number of active model workers [9], we have greater control over system performance. Furthermore, we can deactivate idle models by reducing their worker count to zero and automatically reactivate them when necessary, based on queue backlog demands

In this example, we will discuss a scaling scenario for LLM A. Suppose there are 10 requests waiting in the queue for LLM A, with each worker handling LLM A taking one second to process a single request. We have only one worker for LLM A, and the maximum acceptable latency is set at five seconds. To determine the maximum request backlog for every worker, we divide the maximum acceptable latency by the average time taken to process one request by one worker, resulting in a maximum backlog of five requests per worker.

$$max\_backlog\_per\_worker = max\_allowed\_latency / avg\_time\_for\_1\_request$$

$$max\_backlog = max\_backlog\_per\_worker * current\_num\_workers$$

Therefore, to process the 10 requests within the acceptable latency, we require two LLM workers. Consequently, we add more workers when the request backlog exceeds max_backlog, and we reduce the number of workers when the request backlog is less than max_backlog.

In addition to the ability to horizontally scale workers, it is essential that individual workers are highly efficient to minimize infrastructure requirements for managing a given workload. LLM inference is memory-intensive, with a significant portion of GPU memory bandwidth being used to load model parameters. Therefore, if each inference request is processed individually, a large portion of the GPU capacity of the instance goes unused. To address this issue, we typically gather and process a group of incoming requests within a time window of x milliseconds as a single batch. The next batch is initiated once the previous one is finished. This helps greatly improve GPU utilization when the inference requests take roughly the same amount of time to complete. However, the processing time for LLM inference requests can significantly vary due to the substantial disparity in input and output tokens between requests, leading to some requests finishing before the entire batch and wasting capacity.

The paper "Orca: A Distributed Serving System for Transformer-Based Generative Models" [10] proposes iteration-level scheduling, which enables continuous batching. In continuous batching, once a request is completed, the results are returned, and the

next request is picked up without waiting for the entire batch to finish. Our LLM workers employ this technique when serving Open Source and custom fine-tuned LLMs primarily using the Nvidia TensorRT LLM framework [11]. Nvidia TensorRT LLM offers several optimizations to efficiently serve LLMs with high throughput and low latency. It is important to mention that the architecture permits the use of any framework to serve LLMs, making it simple to switch between different LLM frameworks with minimal modifications.

3.4 Embedding-as-a-service

Embedding-as-a-service is also a microservices architecture that is very similar to the architecture depicted in Fig.4. Embedding-as-a-service accepts a text, embedding model to use and returns embeddings created using the requested model. Similar to LLM-as-a-service, the Request Handler distributes requests to designated model queues. Model workers, in turn, retrieve tasks from these queues and transmit responses back to the user. Embedding-as-a-service employs a batching strategy where we gather and process a group of incoming requests within a time window of x milliseconds as a single batch. The default embedding model served by Embedding-as-a-service is Alibaba-NLP gte-large-en-1.5 [12, 13].

3.5 Rerank-as-a-service

ReRank-as-a-service employs transformer-based models to rearrange all the search results according to their relevance to the augmented query. Rerank-as-a-service follows the same architecture pattern and design decisions as Embedding-as-a-service. The default model served by Rerank-as-a-service is bge-reranker-large [16]

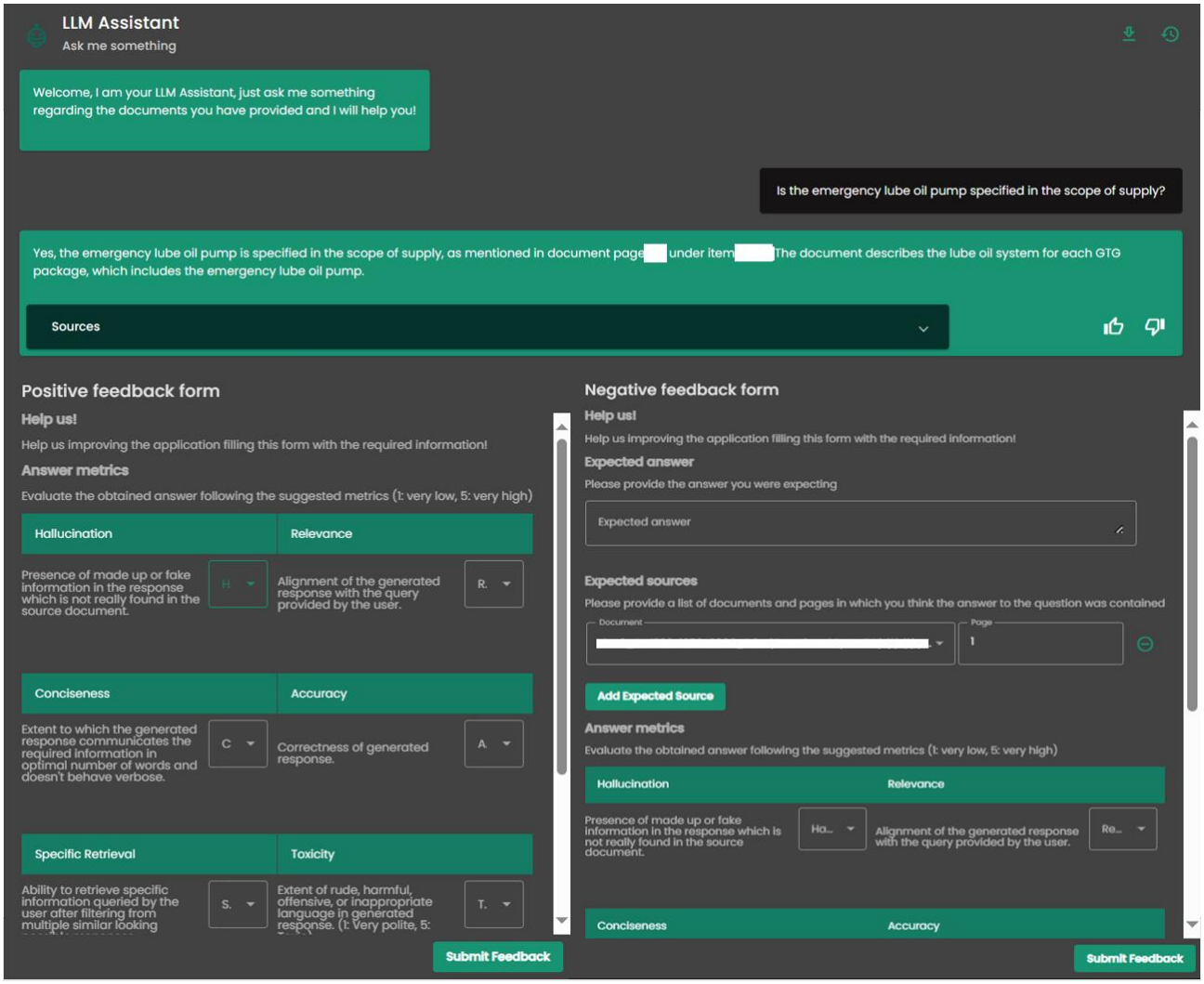

Fig. 5. Feedback collection user interface

4. Feedback Loop

To continually enhance the RAG system, our Application UI includes interfaces for users to submit feedback on the application's responses. Users can express their satisfaction by giving a thumbs up or down to a response. Furthermore, they have the optional ability to provide additional detailed feedback through a comment section and numeric fields. The numeric fields are aligned with our predefined metrics, which include hallucination, conciseness, accuracy, relevance, and toxicity. This feature enables us to better understand the specific issues with the response and helps filter data to be used for fine-tuning. The optional nature of detailed feedback aims to make the process of providing feedback efficient and to encourage more feedback from users.

The accumulated feedback is kept in a database for future analysis. This data can be restructured and formatted, and subsequently combined with publicly available data for the fine-tuning of LLMs. According to the research paper "Leveraging the Regularizing Effect of Mixing Industrial and Open-Source Data to Prevent Overfitting of LLM Fine Tuning" [18], blending technical data with publicly accessible data in a specific ratio can improve fine-tuning outcomes. This data is also used as a source of human evaluation of the current RAG+LLM system.

5. Benchmark

A comprehensive evaluation of new Language Models (LLMs) and Retrieval-Augmented Generation (RAG) system is essential for maintaining consistency, reliability, and delivering optimal user experiences. Traditional evaluation metrics such as BLEU or ROUGE are insufficient for evaluating LLM outputs in a human manner without incurring significant human time and cost. An alternative approach, LLM-as-a-judge, was proposed in the study "Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena" [19]. This method involves utilizing LLMs as judges to evaluate the quality of other LLM outputs.

In our benchmarking process, we use an LLM as a judge to evaluate the performance of our system. Standardized tests are employed to assess the Correctness, Relevance, Groundedness, and Sources of the generated outputs using Prometheus [20] for LLM as a judge. Prometheus [20], a fully open-source LLM that is on par with GPT-4's evaluation capabilities when the appropriate reference materials (reference answer, score rubric) are accompanied. Prometheus is trained on a dataset that consists of 1K fine-grained score rubrics, 20K instructions, and 100K responses and language feedback generated by GPT-4 [20]. The specifics of the dataset, metrics and execution will be discussed in the subsequent sections.

5.1 Dataset

For the benchmark to be a fair indicator we make sure we create the dataset on our real used manuals, technical documents, etc. We analyzed the file documents and created a dataset with the following fields.

1. Query - User question on the documents
2. Sources - The specific document(s) and the corresponding page number(s) that contains the response to the query
3. Context – The exact pieces of information from the sources that contain the response to the query.
4. Ground Truth – Expected response.

The gold standard is to use human generated datasets for benchmarking and training. However, one option instead of relying on human-generated datasets for benchmarking and training is to create synthetic datasets using Language Model Machines (LLMs). A human can be involved in the process to review and audit the results produced by the LLM. References [21,22] explore generating synthetic datasets using LLMs.

5.2 Custom Metrics

To evaluate the RAG system the following metrics are used.

1. **Accuracy Metric:** Accuracy Metric focuses on assessing how correct the generated answer to the user question, referring to a ground truth answer. An answer is deemed correct if is factual consistent and semantically similar to the reference answer; higher score is assigned to an answer that matches the reference answer, conveying the same meaning without contradicting it, while a lower score is assigned to an answer that contradict facts expressed in the reference or conveys a completely different meaning.
2. **Relevance Metric:** Answer Relevance Metric focuses on assessing how pertinent the generated answer is to the given user question. An answer is deemed relevant when it directly and appropriately addresses the original question; higher score is assigned to an answer that directly addresses the key points of the question, provides accurate information and avoids irrelevant details, essentially fulfilling the user's information needs effectively, while a lower score is assigned to answers that are incomplete, not accurate and contain redundant information.
3. **Context Relevance Metric:** Context relevance evaluates the degree to which a context satisfies the information needs expressed in the question. A highly relevant context directly tackles the core aspects of the question, providing crucial information or facilitating further understanding, not introducing irrelevant or tangential details, while a lowly relevant context

has little to no bearing on the question, discuss entirely different topics, offer generic information unrelated to the specific details or is completely out of scope.

4. **Groundness Metric:** Groundedness metric helps detect hallucinations. Groundedness refers to the degree to which an answer leverages supporting information from the context. A highly grounded answer explicitly mentions relevant factual details, entities, or concepts present in the context and demonstrates a clear connection between the information provided and the contextual background while a lowly ground answer provides little to no connection to the context, offering generic information unrelated to the specific details or entities mentioned.
5. **Specific Retrieval Metric:** Specific Retrieval Metric checks if the retrieved source by the RAG contains the sources in benchmark data.
6. **Toxicity:** Toxicity metric checks if the generated answer is Toxic or Rude.
7. **Conciseness Metric:** Conciseness Metric checks if the provided answer is not too verbose.

These metrics are calculated for each query in the dataset where 5 is the max and 0 is the least score.

5.3 Execution

The Benchmarking API is packaged as an independent python library. This RAG Benchmarking API provides a standardized interface for communication between the Benchmarking API and a customized RAG system. To run benchmarks using the API, you need to provide a dataset with the fields specified in section 5.1, along with a RAG system that adheres to the requested interface. The API will then execute the benchmarks and generate an HTML report. This report includes scores for all custom metrics specified in section 5.2, as well as their averages.

6. Results

A pilot study was conducted where a small set of people from different teams were given access to the tool and were encouraged to provide their evaluations, encompassing both favorable and unfavorable comments. As can be seen in Fig. 5. Users were given the option to label a response as positive or negative, while also assigning a metric score ranging from 1 to 5 for 'Hallucination', 'Relevance', 'Accuracy', 'Conciseness', 'Toxicity', and 'Specific Retrieval'. A score of 1 represented the lowest rating, while a score of 5 signified the highest. During our pilot study, we gathered a grand total of 159 feedback, with 140 of these evaluations containing metric-level feedback. If a user only provided feedback on a particular metric and disregarded the others, we assumed that those overlooked metrics were functioning adequately, given their optional nature.

Total Feedback	Positive Feedback	Negative Feedback
159	109	50

	Feedback
Hallucination	4.3/5
Relevance	4.12/5
Conciseness	4.06/5
Accuracy	3.87/5
Toxicity	4.9/5
Specific Retrieval	4.14/5

Note: A score of 5 for hallucination would mean no hallucination and a score of 5 for toxicity would mean very polite.

To examine the scalability of LLM-as-a-service, which is the most infrastructure-heavy component, we conducted benchmarking tests on LLM-as-a-service. 1000 samples from the ‘cnn_dailymail’ dataset was summarized using LLM-as-a-service hosted on a single A100 40GB GPU. The LLM-as-a-service was serving fp16 mistral-7b-v0.2. The following are the results.

Benchmark Dataset Token Count Statistics							
Samples	**Mean Token Count**	**Standard Deviation**	**Min**	**25%**	**50%**	**75%**	**Max**
1001	857.9	403.11	46	543	794	1113	2584

LLM-as-a-service throughput statistics			
Total tokens generated	**Total time**	**Tokens/second**	**Batch Size**

233080	235.5 seconds	989.45	40

7. Limitations and Discussions

The performance of the RAG+LLM system hinges on the ability of the data ingestion pipeline to correctly process the available data. If the quality of the data provided to the LLM by the RAG components is poor, the LLM output will likely be of lower quality as well. Good quality Optical Character Recognition (OCR) systems, table extraction models, and layout parsers tailored to the company's data should be employed to ensure the LLM receives high-quality input for optimal performance. In some scenarios we may note that a single RAG system fails to accommodate two different use cases, in such cases it may be worthwhile to develop custom RAG workflows that leverage supporting services like LLM and embedding.

Keeping up with the latest advancements in Natural Language Processing (NLP) technology can be a complex task. To ensure our RAG system remains competitive, we require reliable and frequently updated evaluation systems that enable us to make prompt adjustments. Regularly updating benchmarking datasets is essential to maintain the relevance of our benchmarks. In addition to keeping the production LLM current, it's crucial to update the Judge LLM as well.

Developing and maintaining microservices can be more expensive than a monolith due to additional complexity created by more moving parts, distributed transactions etc. For smaller-scale deployments, a macro services approach can provide a more cost-effective and manageable solution, preserving scalability while reducing complexity and development costs. An opportunity to further optimize LLM hosting costs is to combine Small Language Models (SLMs) and Large Language Models (LLMs) within the RAG system. For instance, query rewriting could be handled by an SLM, while the final answer could be generated by an LLM with a more extensive context length. When doing this we should consider that the LLM's ability to follow instructions accurately will also impact the output quality.

Finally, it is essential to integrate guardrails to avoid the model to engage toxic and general conversation with the user.

8. Conclusions

RAG+LLM shifts the way industrial companies utilize data, offering locally developed solutions that can be implemented as an API or chatbot. These solutions enhance the simplicity of data access while preserving the confidentiality of highly sensitive information like government data. This improvement facilitates more efficient communication between Field Engineers and Subject Matter Experts, expediting issue resolutions and site activities. For new projects, application and design engineers can leverage the solution to expedite tech specification, standards, and design practices verification. By aligning project proposals with customer requirements from the early stages, rework and change orders during execution are significantly reduced, enhanced customer relationships, and decreased time to serve and cost of quality.

References

1. Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, Meng Wang, Haofen Wang, "Retrieval-Augmented Generation for Large Language Models: A Survey". arXiv:2312.10997
2. N. Kandpal, H. Deng, A. Roberts, E. Wallace, and C. Raffel, “Large language models struggle to learn long-tail knowledge,” in International Conference on Machine Learning. PMLR, 2023, pp. 15 696– 15 707.
3. Y. Zhang, Y. Li, L. Cui, D. Cai, L. Liu, T. Fu, X. Huang, E. Zhao, Y. Zhang, Y. Chen et al., “Siren’s song in the ai ocean: A survey on hallucination in large language models” arXiv:2309.01219, 2023.
4. https://www.searchenginejournal.com/what-is-google-query-expansion-cases-and-examples/7924/?ref=blog.langchain.dev
5. X. Ma, Y. Gong, P. He, H. Zhao, N. Duan., “Query Rewriting for Retrieval-Augmented Large Language Models” arXiv:2305.14283v3.
6. Ivica Kostric, Krisztian Balog., “A Surprisingly Simple yet Effective Multi-Query Rewriting Method for Conversational Passage Retrieval”. arXiv:2406.18960v1 [cs.IR] 27 Jun 2024
7. Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, Percy Liang., "Lost in the Middle: How Language Models Use Long Contexts". arXiv:2307.03172
8. https://www.researchgate.net/publication/220613776_The_Probabilistic_Relevance_Framework_BM25_and_Beyond
9. https://aws.amazon.com/blogs/containers/amazon-elastic-container-service-ecs-auto-scaling-using-custom-metrics/
10. https://www.usenix.org/conference/osdi22/presentation/yu
11. https://nvidia.github.io/TensorRT-LLM/overview.html#about-tensorrt-llm

12. Zehan Li, Xin Zhang, Yanzhao Zhang, Dingkun Long, Pengjun Xie, Meishan Zhang., "Towards General Text Embeddings with Multi-stage Contrastive Learning". arXiv:2308.03281

13. https://huggingface.co/Alibaba-NLP/gte-large-en-v1.5

14. Seungone Kim, Jamin Shin, Yejin Cho, Joel Jang, Shayne Longpre, Hwaran Lee, Sangdoo Yun, Seongjin Shin, Sungdong Kim, James Thorne, Minjoon Seo , ”Prometheus: Inducing Fine-grained Evaluation Capability in Language Models”

15. Hui Huang12, Yingqi Qu2, Jing Liu2, Muyun Yang13, Tiejun Zhao1 1Faculty of Computing, Harbin Institute of Technology, Harbin, China 2Baidu Inc., Beijing, China, “An Empirical Study of LLM-as-a-Judge for LLM Evaluation: Fine-tuned Judge Models are Task-specific Classifiers”

16. https://huggingface.co/BAAI/bge-reranker-large

17. Sujoy Roychowdhury, Sumit Soman, H G Ranjani, Neeraj Gunda, Vansh Chhabra, Sai Krishna Bala, ”Evaluation of RAG Metrics for Question Answering in the Telecom Domain”

18. Mohamed Salah Jebali, Anna Valanzano, Malathi Murugesan, Giacomo Veneri, Giovanni De Magistris., “Leveraging the Regularizing Effect of Mixing Industrial and Open Source Data to Prevent Overfitting of LLM Fine Tuning”

19. Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, Ion Stoica., "Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena". arXiv:2306.05685

20. Seungone Kim, Jamin Shin, Yejin Cho, Joel Jang, Shayne Longpre, Hwaran Lee, Sangdoo Yun, Seongjin Shin, Sungdong Kim, James Thorne, Minjoon Seo., "Prometheus: Inducing Fine-grained Evaluation Capability in Language Models". arXiv:2310.08491

21. Xu Guo and Yiqiang Chen., “Generative AI for Synthetic Data Generation: Methods, Challenges and the Future”. arXiv:2403.04190v1

22. https://www.confident-ai.com/blog/the-definitive-guide-to-synthetic-data-generation-using-llms