Mastering RAG: How To Architect An Enterprise RAG System

Pratik Bhavsar
Galileo Labs

Galileo
Mastering RAG: How To Architect An Enterprise RAG System
Grok the building components of high performance RAG

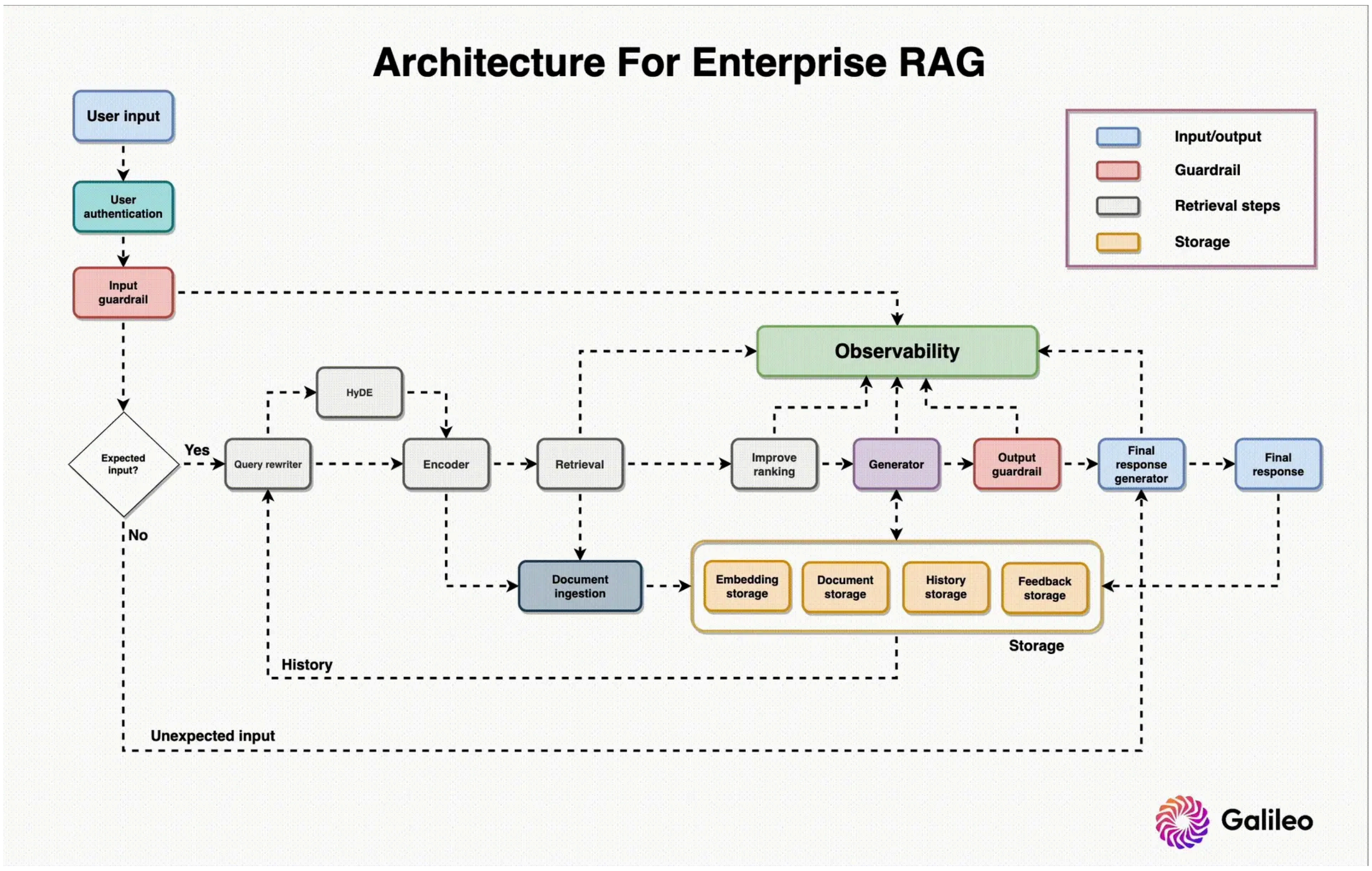
22 min read January 23 2024

Explore our research-backed evaluation metric for RAG – read our paper on Chainpoll.

Welcome back to our Mastering RAG series! Let's roll up our sleeves and delve into the intricate world of building an enterprise RAG system.

architecting a robust enterprise-level solution is often a mystery.

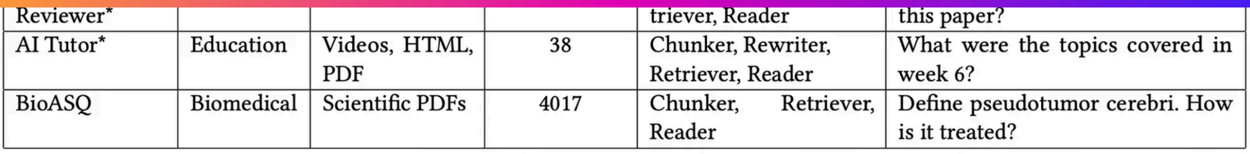
But this blog isn't just a theoretical journey; it's a practical guide to help you take action! From the importance of guardrails in ensuring security to the impact of query rewriting on the user experience, we're here to provide actionable insights and real-world examples. Whether you’re a seasoned developer or a technical leader steering the ship, buckle up for a deep dive into the intricate world of cutting-edge enterprise RAG!



Before touching on RAG architecture, I want to share a recent study on common failure points when building RAG systems. Researchers analyzed three case studies across unique domains and found seven common RAG failure points.

Challenges in Building RAG Systems

Case studies



Reviewer*				triever, Reader	this paper?
AI Tutor*	Education	Videos, HTML, PDF	38	Chunker, Rewriter, Retriever, Reader	What were the topics covered in week 6?
BioASQ	Biomedical	Scientific PDFs	4017	Chunker, Retriever, Reader	Define pseudotumor cerebri. How is it treated?

Table 1: A summary of the RAG case studies presented in this paper. Case studies marked with a * are running systems currently in use.

Cognitive reviewer

Cognitive reviewer is a RAG system designed to assist researchers in analyzing scientific documents. Researchers define a research question or objective and upload a collection of related research papers. The system then ranks all documents based on the stated objective for manual review by the researcher. Additionally, researchers can pose questions directly against the entire document set.

AI tutor

AI Tutor, another RAG system, enables students to ask questions about a unit and receive answers sourced from learning content. Students can verify answers by accessing a sources list. Integrated into Deakin's learning management system, AI Tutor indexes all content, including PDFs, videos, and text documents. The system transcribes videos using the Whisper deep learning model before chunking. The RAG pipeline incorporates a rewriter for query generalization, and a chat interface utilizes past dialogues for context in each question.

Biomedical Q&A

In the Biomedical Question and Answer case study, a RAG system was created using the BioASQ dataset, containing questions, document links, and answers. The dataset, prepared by biomedical experts, includes domain-specific question-answer pairs. The answers to questions were either yes/no, text summarisation, factoid, or list.

7 Failure Points of RAG Systems

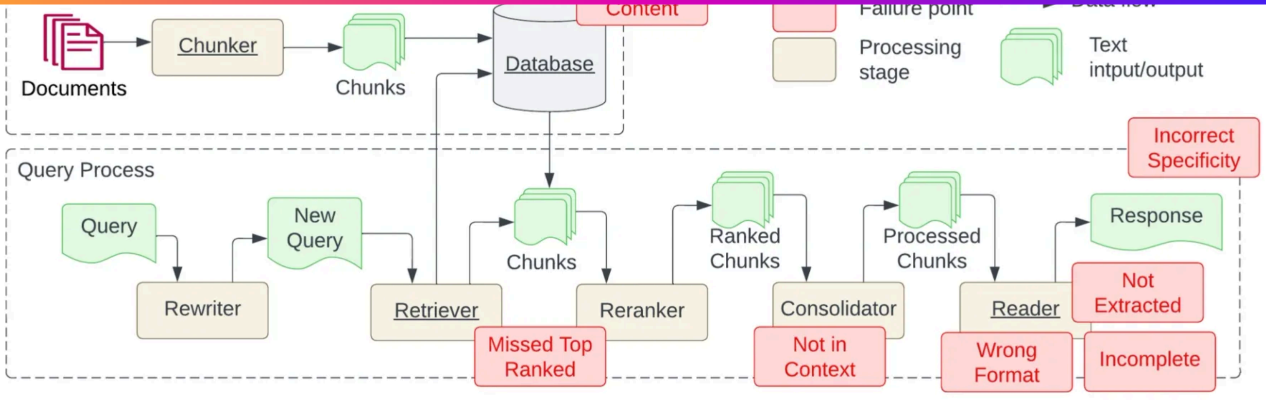


Figure 1: Indexing and Query processes required for creating a Retrieval Augmented Generation (RAG) system. The indexing process is typically done at development time and queries at runtime. Failure points identified in this study are shown in red boxes. All required stages are underlined. Figure expanded from [19].

Through these case studies, they identified seven failure points that often arise when engineering a RAG system.

Missing content (FP1)

A question is posed that cannot be answered with the available documents. In the ideal scenario, the RAG system responds with a message like "Sorry, I don't know." However, for questions related to content without clear answers, the system might be misled into providing a response.

Missed the top ranked documents (FP2)

The answer to a question is present in the document, but did not rank highly enough to be included in the results returned to the user. While all documents are theoretically ranked and utilized in subsequent steps, in practice only the top K documents are returned, with K being a value selected based on performance.

Not in context - consolidation strategy limitations (FP3)

Documents containing the answer are retrieved from the database but fail to make it into the context for generating a response. This occurs when a substantial number of documents are returned, leading to a consolidation process where the relevant answer retrieval is hindered.

Not extracted (FP4)

information. This typically happens when there is excessive noise or conflicting information in the context.

Wrong format (FP5)

The question involves extracting information in a specific format, such as a table or list, and the model disregards the instruction.

Incorrect specificity (FP6)

The response includes the answer but lacks the required specificity or is overly specific, failing to address the user's needs. This occurs when RAG system designers have a predetermined outcome for a given question, such as teachers seeking educational content. In such cases, specific educational content should be provided along with answers. Incorrect specificity also arises when users are unsure how to phrase a question and are too general.

Incomplete (FP7)

Incomplete answers are accurate but lack some information, even though that information was present in the context and available for extraction. For instance, a question like "What are the key points covered in documents A, B, and C?" would be better approached by asking these questions separately.

We will keep these lessons in mind while architecting our enterprise RAG system.

How to Build an Enterprise RAG System

Now that we've established common problems faced when designing a RAG system, let's go through the design needs and roles of each component, along with best practices for building them. The RAG system architecture diagram above offers context on where and how each component is used.

User authentication

Where it all starts – the first component in our system! Before the user can even start interacting with the chatbot, we need to authenticate the user for various reasons. Authentication helps with security and personalization, which is a must for enterprise systems.

Access Control

Authentication ensures that only authorized users gain access to the system. It helps control who can interact with the system and which actions they are allowed to perform.

Data Security

Protecting sensitive data is paramount. User authentication prevents unauthorized individuals from accessing confidential information, preventing data breaches and unauthorized data manipulation.

User Privacy

Authentication helps maintain user privacy by ensuring that only the intended user can access their personal information and account details. This is crucial for building trust with users.

Legal Compliance

organizations to implement proper user authentication to protect user data and privacy. Adhering to these regulations helps avoid legal issues and potential penalties.

Accountability

Authentication ensures accountability by tying actions within the system to specific user accounts. This is essential for auditing and tracking user activities, helping to identify and address any security incidents or suspicious behavior.

Personalization and Customization

Authentication allows systems to recognize individual users, enabling personalization and customization of user experiences. This can include tailored content, preferences, and settings.

Services like AWS Cognito or Firebase Authentication can help you easily add user sign-up and authentication to mobile and web apps.

Input guardrail

It’s essential to prevent user inputs that can be harmful or contain private information. Recent studies have shown it's easy to jailbreak LLMs. Here’s where input guardrails come in. Let's have a look at different scenarios for which we need guardrails.

Anonymization

Input guardrails can anonymize or redact personally identifiable information (PII) such as names, addresses, or contact details. This helps to protect privacy and prevent malicious attempts to disclose sensitive information.

Restrict substrings

Prohibiting certain substrings or patterns that could be exploited for SQL injection, cross-site scripting (XSS), or other injection attacks prevents security vulnerabilities or unwanted behaviors.

In order to restrict discussions or inputs related to specific topics that may be inappropriate, offensive, or violate community guidelines, it’s important to filter out content that involves hate speech, discrimination, or explicit material.

Restrict code

It’s essential to prevent the injection of executable code that could compromise system security or lead to code injection attacks.

Restrict language

Verify that text inputs are in the correct language or script, preventing potential misinterpretations or errors in processing.

Detect prompt injection

Mitigate attempts to inject misleading or harmful prompts that may manipulate the system or influence the behavior of LLMs in unintended ways.

Limit tokens

Enforcing a maximum token or character limit for user inputs helps avoid resource exhaustion and prevents denial-of-service (DoS) attacks.

Detect toxicity

Implement toxicity filters to identify and block inputs that contain harmful or abusive language.

To safeguard your RAG system against these scenarios, you can leverage Llama Guard by Meta. You can host it yourself or use a managed service like Sagemaker. However, do not expect it to be perfect at detecting toxicity.

Query rewriter

Once the query passes the input guardrail, we send it to the query rewriter. Sometimes, user queries can be vague or require context to understand the involves transforming user queries to enhance clarity, precision, and relevance. Let's go through some of the most popular techniques.

Rewrite based on history

In this method, the system leverages the user's query history to understand the context of the conversation and enhance subsequent queries. Let's use an example of a credit card inquiry.

Query History:

"How many credit cards do you have?"

"Are there any yearly fees for platinum and gold credit cards?"

"Compare features of both."

We must identify the context evolution based on the user's query history, discern the user's intent and relationship between queries, and generate a query that aligns with the evolving context.

Rewritten Query: "Compare features of platinum and gold credit cards."

Create subqueries

Complex queries can be difficult to answer due to retrieval issues. To simplify the task, queries are broken down into more specific subqueries. This helps to retrieve the right context needed for generating the answer. LlamaIndex refers to this as a sub question query engine.

Given the query "Compare features of platinum and gold credit card," the system generates subqueries for each card that focus on individual entities mentioned in the original query.

Rewritten Subqueries:

1. "What are the features of platinum credit cards?"
2. "What are the features of gold credit cards?"

Create similar queries

queries based on user input. This is to overcome the limitations of the retrieval in semantic or lexical matching.

If the user asks about credit card features, the system generates related queries. Use synonyms, related terms, or domain-specific knowledge to create queries that align with the user's intent.

Generated Similar Query:

“I want to know about platinum credit cards” → "Tell me about the benefits of platinum credit cards."

Encoder

Once we have the original and rewritten queries, we encode them into vectors (a list of numbers) for retrieval. ***Choosing an encoder is probably the most important decision in building your RAG system.*** Let’s explore why and the factors to consider when choosing your text encoder.

Leveraging MTEB benchmarks

For a comprehensive assessment of encoder capabilities, the go-to source is the Massive Text Embedding Benchmark (MTEB). This benchmark allows for a nuanced selection of encoders based on vector dimension, average retrieval performance, and model size. While the MTEB provides valuable insights, it's essential to approach the results with a degree of skepticism, as there is no one-size-fits-all evaluation benchmark, and the specifics of the models' training data may not be fully disclosed.

MTEB not only provides insights into the performance of popular embeddings such as OpenAI, Cohere, and Voyager, but also reveals that certain open-source models exhibit close performance levels. However, it's important to note that these results offer a general overview and may not precisely indicate how well these embeddings will perform within the specific context of your domain. Therefore, **it is imperative to perform a thorough evaluation on your dataset before making a final selection**, emphasizing the significance of custom evaluation methodologies.

Encoders may not consistently deliver optimal performance, especially when handling sensitive information. Custom evaluation methods become crucial in such scenarios. Here are three approaches to performing custom evaluations.

Evaluation by annotation

Generate a dedicated dataset and set up annotations to obtain gold labels. After annotation, leverage retrieval metrics like Mean Reciprocal Rank (MRR) and Normalized Discounted Cumulative Gain (NDCG) to assess the performance of different encoders quantitatively.

Evaluation by model

Follow a data generation process similar to the annotation approach, but use an LLM or a cross-encoder as the evaluator. This allows the establishment of a relative ranking among all encoders. Subsequently, manual assessment of the top three encoders can yield precise performance metrics.

Evaluation by clustering

Employ diverse clustering techniques and analyze the coverage (quantity of data clustered) at distinct Silhouette scores, indicating vector similarity within clusters. Experiment with algorithms like HDBSCAN, adjusting their parameters for optimal performance selection. This clustering-based evaluation provides valuable insights into the distribution and grouping of data points, aiding in selecting encoders that align with specific metrics.

Consideration Of Selecting A Text Encoder

When choosing your encoder, you'll need to decide between a private encoder or a public encoder. You might be tempted to use a private encoder due to their ease-of-use, but there are specific tradeoffs that require consideration between the two options. It's an important decision that will decide the performance and latency of your system.

Querying cost

Ensuring a smooth user experience in semantic search relies on the high availability of the embedding API service. OpenAI and similar providers offer latency needs. Smaller models (up to 110M parameters) can be hosted with a CPU instance, while larger models may demand GPU serving to meet latency requirements.

Indexing cost

Setting up semantic search involves indexing documents, incurring a non-trivial cost. As indexing and querying share the same encoder, the indexing cost hinges on the chosen encoder service. To facilitate service resets or reindexing onto an alternative vector database, it's advisable to store embeddings separately. Neglecting this step would necessitate recalculating identical embeddings.

Storage Cost

For applications indexing millions of vectors, Vector DB storage cost is a crucial consideration. Storage cost scales linearly with dimension, and OpenAI's embeddings in 1526 dimensions incur the maximum storage cost. To estimate storage cost, calculate average units (phrase or sentence) per doc and extrapolate.

Language Support

In order to support your non-English language either use a multi-lingual encoder or use a translation system along with an English encoder.

Search latency

The latency of semantic search grows linearly with the dimension of the embeddings. Opting for lower-dimensional embeddings is preferable to minimize latency.

Privacy

Stringent data privacy requirements in sensitive domains like finance and healthcare may render services like OpenAI less viable.

Document ingestion

data. During the indexing process each document is split into smaller chunks that are converted into an embedding using an embedding model. The original chunk and the embedding are then indexed in a database. Let's look at the components of the document ingestion system.

Document parser

The document parser takes a central role in actively extracting structured information from diverse document formats, with a particular focus on format handling. This includes, but is not limited to, parsing PDFs that may contain images and tables.

Document formats

The document parser must demonstrate proficiency in handling a variety of document formats, such as PDF, Word, Excel, and others, ensuring adaptability in document processing. This involves identifying and managing embedded content, such as hyperlinks, multimedia elements, or annotations, to provide a comprehensive representation of the document.

Table recognition

Recognizing and extracting data from tables within documents is imperative for maintaining the structure of information, especially in reports or research papers. The extraction of metadata related to tables, including headers, row and column information, enhances the comprehension of the document's organizational structure. Models such as Table Transformer can be useful for the task.

Image recognition

OCR is applied to images within documents to actively recognize and extract text, making it accessible for indexing and subsequent retrieval.

Metadata extraction

Metadata refers to additional information about the document that is not part of its main content. It includes details such as author, creation date, document type, keywords, etc. Metadata provides valuable context and helps in organizing documents and to improve the relevance of search results by considering and indexed with the docs as special fields.

Chunker

How you decide to tokenize (break) longform text can decide the quality of your embeddings and the performance of your search system. If chunks are too small, certain questions cannot be answered; if the chunks are too long, then the answers include generated noise. You can exploit summarisation techniques to reduce noise, text size, encoding cost and storage cost.

Chunking is an important yet underrated topic. It can require domain expertise similar to feature engineering. For example, chunking for python codebases might be done with prefixes like def/class. Read our blog for a deeper dive into chunking.

Chunking Techniques For RAG

Indexer

The indexer is – you guessed it – responsible for creating an index of the documents, which serves as a structured data structure (say that 3 times fast...). The indexer facilitates efficient search and retrieval operations. **Efficient indexing is crucial for quick and accurate document retrieval.** It involves creating an index and adding, updating, or deleting documents.

The indexer, being a critical component of a RAG system, faces various challenges and issues that can impact the overall efficiency and performance of the system.

Scalability issues

As the volume of documents grows, maintaining efficient and fast indexing becomes challenging. Scalability issues may arise when the system struggles to handle an increasing number of documents, leading to slower indexing and retrieval times.

Real-time index updates

Keeping the index up-to-date in real-time can be challenging, especially in systems where documents are frequently added, updated, or deleted. Ensuring that live APIs and real-time indexing mechanisms operate seamlessly without compromising system performance is a persistent challenge.

Consistency and atomicity

Achieving consistency and atomicity in the face of concurrent document updates or modifications can be complex. Ensuring that updates to the index maintain data integrity, even in the presence of simultaneous changes, requires careful design and implementation.

Optimizing storage space

Indexing large volumes of documents may lead to considerable storage requirements. Optimizing storage space while ensuring that the index remains accessible and responsive is an ongoing challenge, especially in scenarios where storage costs are a concern.

Security and access control

unauthorized modifications to the index is crucial. Ensuring that only authorized users or processes can perform CRUD operations helps protect the integrity of the document repository.

Monitoring and maintenance

Regularly monitoring the health and performance of the indexer is essential. Detecting issues, such as indexing failures, resource bottlenecks, or outdated indexes, requires robust monitoring and maintenance procedures to ensure the system operates smoothly over time.

These are some difficult but well known software engineering challenges which can be tackled by following good software design practices.

Data storage

Since we are dealing with a variety of data we need dedicated storage for each of them. It's critical to understand the different considerations for every storage type and specific use cases of each.

Embeddings

Database type: SQL/NoSQL

Storing document embeddings separately allows for swift reindexing without recalculating embeddings for the entire document corpus. Additionally, embedding storage acts as a backup, ensuring the preservation of critical information even in the event of system failures or updates.

Documents

Database type: NoSQL

Document storage in its raw format is essential for persistent storage. This raw format serves as the foundation for various processing stages, such as indexing, parsing, and retrieval. It also provides flexibility for future system enhancements, as the original documents remain intact and can be reprocessed as needed.

Database type: NoSQL

The storage of chat history is imperative for supporting the conversational aspect of the RAG system. Chat history storage allows the system to recall previous user queries, responses, and preferences, enabling it to adapt and tailor future interactions based on the user's unique context. This historical data is a valuable resource for improving the ML system by leveraging it for research.

User feedback

Database type: NoSQL/SQL

User feedback is systematically collected through various interaction mechanisms within the RAG application. In most LLM systems, users can provide feedback using thumbs-up/thumbs-down, star ratings and text feedback. This array of user insights serves as a valuable repository, encapsulating user experiences and perceptions, forming the basis for ongoing system enhancements.

Vector database

The vector database powering the semantic search is a crucial retrieval component of RAG. However, selecting this component appropriately is vital to avoid potential issues. Several vector database factors need to be considered in the selection process. Let’s go over some of them.

Recall vs. Latency

Optimizing for recall (percentage of relevant results) versus latency (time to return results) is a trade-off in vector databases. Different indexes like Flat, HNSW (Hierarchical Navigable Small World), PQ (Product quantization), ANNOY, and DiskANN make varying trade-offs between speed and recall. Conduct benchmark studies on your data and queries to make an informed decision.

Cost

storage and query volume. This model is suitable for organizations with substantial data, avoiding infrastructure costs. Key considerations include evaluating dataset growth, the team's capability, data sensitivity, and understanding the cost implications of managed cloud solutions.

On the other side, self-hosting provides organizations with more control over their infrastructure and potentially lower costs. However, it comes with the responsibility of managing and maintaining the infrastructure, including considerations for scalability, security, and updates.

Insertion speed vs. Query speed

Balancing insertion speed and query speed is vital. Look for vendors that can handle streaming use cases with high insertion speed requirements. However, for most organizations, prioritizing querying speed is more relevant. Evaluate the vector insertion speed query latency at peak loads to make an informed decision.

In-memory vs. On-disk index storage

Choosing between in-memory and on-disk storage involves speed and cost trade-offs. While in-memory storage offers high speed, some use cases require storing vectors larger than memory. Techniques like memory-mapped files allow scaling vector storage without compromising search speed. New indexes like Vamana in DiskANN promise efficient out-of-memory indexing.

Full-Text search vs. Vector Hybrid search

Source: https://www.pinecone.io/learn/hybrid-search-intro/

Vector search alone may not be optimal for enterprise-level applications. On the other hand, hybrid search, which integrates both dense and sparse dense and sparse elements is adjustable through a parameter known as alpha in Pinecone, Weaviate & Elasticsearch.

Filtering

Real-world search queries often involve filtering on metadata attributes. Pre-filtered search, although seemingly natural, can lead to missing relevant results. Post-filtered search may have issues if the filtered attribute is a small fraction of the dataset. Custom-filtered search, like Weaviate, combines pre-filtering with effective semantic search using inverted index shards alongside HNSW index shards.

Techniques for improving retrieval

Recent research has shown that LLMs can be easily distracted by irrelevant context and having a lot of context (topK retrieved docs) can lead to missing out of certain context due to the attention patterns of LLMs. Therefore it is crucial to improve retrieval with relevant and diverse documents. Let's look at some of the proven techniques for improving retrieval.

Hypothetical document embeddings (HyDE)

We can use the HyDE technique to tackle the problem of poor retrieval performance, especially when dealing with short or mismatched queries that can make finding information difficult. HyDE takes a unique approach by using hypothetical documents created by models like GPT. These hypothetical documents capture important patterns but might have made-up or incorrect details. A smart text encoder then turns this hypothetical document into a vector embedding. This embedding helps find similar real documents in the collection better than embedding of query.

Experiments show that HyDE works better than other advanced methods, making it a useful tool to boost the performance of RAG systems.

Query routing

queries to the most relevant index for efficient retrieval. This approach streamlines the search process by ensuring that each query is directed to the appropriate index, optimizing the accuracy and speed of information retrieval.

In the context of enterprise search, where data is indexed from diverse sources such as technical documents, product documentation, tasks, and code repositories, query routing becomes a powerful tool. For instance, if a user is searching for information related to a specific product feature, the query can be intelligently routed to the index containing product documentation, enhancing the precision of search results.

Reranker

When retrieval from the encoder falls short of delivering optimal quality, a reranker is used to enhance the document ranking. Utilizing open-source encoder-only transformers like BGE-large in a cross-encoder setup has become a common practice. Recent decoder-only approaches, such as RankVicuna, RankGPT, and RankZephyr have further boosted reranker performance.

Introducing a reranker has benefits, reducing LLM hallucinations in responses and improving the system's out-of-domain generalization. However, it comes with drawbacks. Sophisticated rerankers may increase latency due to computational overhead, impacting real-time applications. Additionally, deploying advanced rerankers can be resource-intensive, demanding careful consideration of the balance between performance gains and resource utilization.

Maximal Marginal Relevance (MMR)

MMR is a method designed to enhance the diversity of retrieved items in response to a query, avoiding redundancy. Rather than focusing solely on retrieving the most relevant items, MMR achieves a balance between relevance and diversity. It’s like introducing a friend to people at a party. Initially, it identifies the most matching person based on the friend's preferences. Then, it seeks someone slightly different. This process continues until the desired number of introductions is achieved. MMR ensures a more diverse and relevant set of items is presented, minimizing redundancy.

The original MMR

Autocut

The autocut feature from Weaviate, is designed to limit the number of search results returned by detecting groups of objects with close scores. It works by analyzing the scores of the search results and identifying significant jumps in these values, which can indicate a transition from highly relevant to less relevant results.

For example, consider a search that returns objects with these distance values:

[0.1899, 0.1901, 0.191, 0.21, 0.215, 0.23].

Autocut returns the following:

- autocut: 1: [0.1899, 0.1901, 0.191]
- autocut: 2: [0.1899, 0.1901, 0.191, 0.21, 0.215]
- autocut: 3: [0.1899, 0.1901, 0.191, 0.21, 0.215, 0.23]

Recursive retrieval

Source: https://youtu.be/TRjq7t2Ms5I?si=D0z5sHKW4SMqMgSG&t=742

Recursive retrieval, aka the small-to-big retrieval technique, embeds smaller chunks for retrieval while returning larger parent context for the language model's synthesis. Smaller text chunks contribute to more accurate retrieval, while larger chunks provide richer contextual information for the language model. This sequential process optimizes the accuracy of retrieval by initially focusing on smaller, more information-dense units, which are then efficiently linked to their broader contextual parent chunks for synthesis.

Sentence window retrieval

The retrieval process fetches a single sentence and returns a window of text around that particular sentence. Sentence window retrieval ensures that the information retrieved is not only accurate but also contextually relevant, offering comprehensive information around the main sentence.

Generator

Now that we've discussed all retrieval components let's talk about the generator. It requires careful considerations and trade-offs mainly between self-hosted inference deployment and private API services. This is a large topic in itself, and I will touch on it briefly to avoid overwhelming you.

API considerations

When evaluating an API server for LLMs, it is crucial to prioritize features that ensure seamless integration and robust performance. A well-designed API should considerations such as production readiness, security, and hallucination detection. Notably, the TGI server from HuggingFace exemplifies a comprehensive set of features that embody these principles. Let's understand some of the most popular features needed in a LLM server.

Performance

An efficient API must prioritize performance to cater to diverse user needs. Tensor parallelism stands out as a feature that facilitates faster inference on multiple GPUs, enhancing the overall processing speed. Additionally, continuous batching of incoming requests ensures an increased total throughput, contributing to a more responsive and scalable system. The inclusion of quantization, specifically with bitsandbytes and GPT-Q, further optimizes the API for enhanced efficiency across various use cases. The ability to utilize optimized transformers code ensures seamless inference on the most popular architectures.

Generation quality enhancers

To elevate the quality of generation, the API should incorporate features that can transform the output. The logits processor, encompassing temperature scaling, top-p, top-k, and repetition penalty, allows users to customize the output according to their preferences. Moreover, a stop sequences provides control over the generation, enabling users to manage and refine the content generation process. Log probabilities, crucial for hallucination detection, serve as an additional layer of refinement, ensuring that the generated output aligns with the intended context and avoids misleading information.

Security

The security of an API is paramount, particularly when dealing with LLMs and enterprise use cases. Safetensors weight loading is an essential feature, contributing to the secure deployment of models by preventing unauthorized tampering with model parameters. Furthermore, the inclusion of watermarking adds an extra layer of security, enabling traceability and accountability in the usage of LLMs.

User experience

seamless interaction. Utilizing Server-Sent Events (SSE) for token streaming enhances the real-time responsiveness of the API, providing users with a smoother and more interactive experience. This ensures that users can receive generated content incrementally, improving the overall engagement and usability of the LLM.

Self-hosted inference

Self-hosted inference involves deploying LLMs on servers provided by cloud service providers like AWS, GCP, or Azure. The choice of servers, such as TGI, Ray, or FastAPI, is a critical decision that directly impacts the system's performance and cost. Considerations include computational efficiency, ease of deployment, and compatibility with the selected LLM.

Measuring LLM inference performance is crucial, and leaderboards like Anyscale's LLMPerf Leaderboard are invaluable. It ranks inference providers based on key performance metrics, including time to first token (TTFT), inter-token latency (ITL), and success rate. Load tests and correctness tests are vital for evaluating different characteristics of hosted models.

In new approaches, Predibase's LoRAX introduces an innovative way to serve fine-tuned LLMs efficiently. It addresses the challenge of serving multiple fine-tuned models using shared GPU resources.

Private API services

LLM API services by companies like OpenAI, Fireworks, Anyscale, Replicate, Mistral, Perplexity and Together, present alternative deployment approaches. It’s essential to understand their features, pricing models, and LLM performance metrics. For instance, OpenAI's token-based pricing, with distinctions between input and output tokens, can significantly impact the overall cost of using the API. When comparing the cost of private API services versus self-hosted LLMs, you must consider factors such as GPU costs, utilization, and scalability issues. For some, rate limits can be a limiting factor (ha!).

Prompting techniques for improving RAG

Mastering RAG series we did a deep dive into the top 5 most effective ones. Many of these new techniques surpass the performance of CoT (Chain-of-Thought). You can also combine them to minimize hallucinations.

LLM prompting techniques for RAG

Output guardrail

The output guardrail functions similarly to its input counterpart but is specifically tailored to detect issues in the generated output. It focuses on identifying hallucinations, competitor mentions, and potential brand damage as part of RAG evaluation. The goal is to prevent generating inaccurate or ethically questionable information that may not align with the brand's values. By actively monitoring and analyzing the output, this guardrail ensures that the generated content remains factually accurate, ethically sound, and consistent with the brand's guidelines.

Here is an example of a response that can damage an enterprise brand but would be blocked by an appropriate output guardrail:

Example of a toxic output

User feedback

Once an output is generated and served, it is helpful to get both positive or negative feedback from users. User feedback can be very helpful for improving the flywheel of the RAG system, which is a continuous journey rather than a one-time endeavor. This entails not only the routine execution of automated tasks like reindexing and experiment reruns but also a systematic approach to integrate user insights for substantial system enhancements.

The most impactful lever for system improvement lies in the active remediation of issues within the underlying data. RAG systems should include an iterative workflow for handling user feedback and driving continuous improvement.

User interaction and feedback collection

Users interact with the RAG application and utilize features such as 👍 / 👎 or star ratings to provide feedback. This diverse set of feedback mechanisms serves as a valuable repository of user experiences and perceptions regarding the system's performance.

Issue identification and diagnostic inspection

After collecting feedback, the team can conduct a comprehensive analysis to identify queries that may be underperforming. This involves inspecting retrieved retrieval, generation, or the underlying data source.

Data improvement strategies

Once issues are identified, particularly those rooted in the data itself, the team can strategically devise plans to enhance data quality. This may involve rectifying incomplete information or restructuring poorly organized content.

Evaluation and testing protocols

After implementing data improvements, the system must undergo rigorous evaluation on previously underperforming queries. Insights gained from these evaluations can then be methodically integrated into the test suite, ensuring ongoing scrutiny and refinement based on real-world interactions.

By actively engaging users in this comprehensive feedback loop, the RAG system not only addresses issues identified through automated processes but also harnesses the richness of user experiences.

Observability

Building a RAG system does not end with putting the system into production. Even with robust guardrails and high-quality data for fine-tuning, models require constant monitoring once in production. Generative AI apps, in addition to standard metrics like latency and cost, need specific LLM observability to detect and correct issues such as hallucinations, out-of-domain queries, and chain failures. Now let's have a look at the pillars of LLM observability.

Prompt analysis and optimization

Identify prompt-related problems and iterate using live production data to identify and address issues like hallucinations using robust evaluation mechanisms.

Traceability in LLM applications

Capture LLM traces from common frameworks like Langchain and LlamaIndex to debug prompts and steps.

Troubleshoot and evaluate RAG parameters to optimize retrieval processes critical to LLM performance.

Alerting

Get alerts if system behavior diverges from the expected, such as increased errors, high latency and hallucinations.

First and foremost, real-time monitoring is essential to observe the performance, behavior, and overall health of your applications in a production environment. Keep a vigilant eye on SLA compliance and set up alerts to promptly address any deviations. Effectively track the costs associated with running LLM applications by analyzing usage patterns and resource consumption to help you in cost optimization.

Galileo's LLM Studio offers purpose-built LLM observability to alert and take immediate corrective action before user complaints proactively. Galileo's guardrail metrics are designed to monitor the quality and safety of your models, covering aspects like groundedness, uncertainty, factuality, tone, toxicity, PII, and more. These metrics, previously utilized during evaluation and experimentation, can now be seamlessly integrated into the monitoring phase.

Additionally, you have the flexibility to register custom metrics to tailor the monitoring process to your specific needs. Leverage insights and alerts generated from monitoring data to stay informed about potential issues, anomalies, or areas for improvement that demand attention. This comprehensive approach ensures that your LLM applications operate efficiently and securely in real-world scenarios.

Caching

For companies operating at scale, cost can become a hindrance. Caching is a great way to save money in such cases. Caching involves the storage of prompts and their corresponding responses in a database, enabling their retrieval for subsequent use. This strategic caching mechanism empowers LLM applications to expedite and economize responses with three distinct advantages.

Caching contributes to faster and more cost-effective inference during production. Certain queries can achieve near-zero latency by leveraging cached responses, streamlining the user experience.

Accelerated development cycles

In the development phase, caching proves to be a boon as it eliminates the need to invoke the API for identical prompts repeatedly. This results in faster and more economical development cycles.

Data storage

The existence of a comprehensive database storing all prompts simplifies the fine-tuning process for LLMs. Utilizing the stored prompt-response pairs streamlines the optimization of the model based on accumulated data.

If you are serious, you can leverage GPTCache to implement caching for exact and similar matches. It offers valuable metrics such as cache hit ratio, latency, and recall, which provide insights into the cache's performance, enabling continuous refinement to ensure optimal efficiency.

Multi-tenancy

SaaS software often has multiple tenants, balancing simplicity and privacy. For multi-tenancy in RAG systems, the goal is to build a system that not only finds information effectively but also respects each user's data limits. In simpler terms, every user's interaction with the system is separate, ensuring the system only looks at and uses the information that is meant for that user.

One of the simple ways to build multi-tenancy is by using metadata. When we add documents to the system, we include specific user details in the metadata. This way each document is tied to a particular user. When someone searches, the system uses this metadata to filter and only show documents related to that user. It then does a smart search to find the most important information for that user. This approach stops private information from being mixed up between different users, keeping each person's data safe and private.

Conclusion

It should be clear that building a robust and scalable enterprise RAG system involves a careful orchestration of interconnected components. From user authentication to input guardrails, query rewriting, encoding, document ingestion, and retrieval components like vector databases and generators, every step plays a crucial role in shaping the system's performance.

In the ever-evolving landscape of RAG systems, we hope this practical guide helps empower developers and leaders with actionable insights!

Galileo GenAI Studio is the leading platform for rapid evaluation, experimentation and observability for teams building LLM powered applications. It is powered by a suite of metrics to identify and mitigate hallucinations. Join 1000s of developers building apps powered by LLMs and get early access!

Read our latest LLM Hallucination Index: RAG Special - See which LLM comes out on top!

Working with Natural Language Processing?

Read about Galileo's NLP Studio

Natural Language Processing
Learn more

Galileo

PRODUCTS RESOURCES COMPANY

GenAI StudioDocs Careers

Blog Privacy Policy

Request a Demo

Subscribe to our newsletter

Email*

Submit

Contact us at info@rungalileo.io