Use These 8 Practices to Improve RAG Systems

30 August 2024 - ID G00807995 - 55 min read

By: Joe Antelmi

Initiatives:Analytics and Artificial Intelligence for Technical Professionals; Architect, Implement and Scale Data and Analytics Solutions; Generative AI Resource Center

Despite the popularity of retrieval-augmented generation, organizations are struggling to optimize large language model applications based on RAG. The best practices in this research help data and analytics architects avoid common pitfalls and deliver production-grade RAG solutions.

Overview

Key Findings

- Although the most common implementations of retrieval-augmented generation (RAG) involve some variation of Q&A over internal data, every RAG system has unique features. Different use cases, questions, data and specialized requirements can significantly complicate RAG system development.
- It is very difficult to build a RAG system that can always respond to diverse, challenging prompts with the right context from diverse, complex data. Building the right set of evaluations and guardrails for RAG systems is therefore crucial, but challenging.
- Useful and valuable RAG systems require many components and integrated processes, making them hard to optimize — and sustain — with high quality and performance. AI architects and engineers struggle to satisfy functional and nonfunctional requirements of RAG-based applications, including component selection, data extraction, search, prompting and quality validation.

Recommendations

As an AI architect or AI engineer looking to improve the quality of RAG-based solutions, you must:

- Gather requirements, and focus on identifying and integrating contextual information sources to solve known business problems rather than hypothetical ones.
- Avoid building a RAG solution unless you are certain that it is worth the effort versus simpler alternatives, such as deploying an LLM-based application for simple summarization, translation or content generation use cases. If you require a RAG solution, build a specialized system focused on a specific use case and corpus of data, as it will be far easier to evaluate and optimize.
- Optimize the solution by focusing on the simplest and most impactful components and stages of the RAG system. For example, concentrate on better information extraction, more intelligent chunking and more intelligent retrieval, as well as your user data and system flywheels, prompt templates, guardrails, and evaluations.

Analysis

Definition of Retrieval-Augmented Generation

Retrieval-augmented generation (RAG) is a pattern for building large language model (LLM) prototypes with access to internal, external and/or licensed context. Properly implemented, RAG systems unify information retrieval and generation to give LLMs the right context to provide relevant and correct answers.

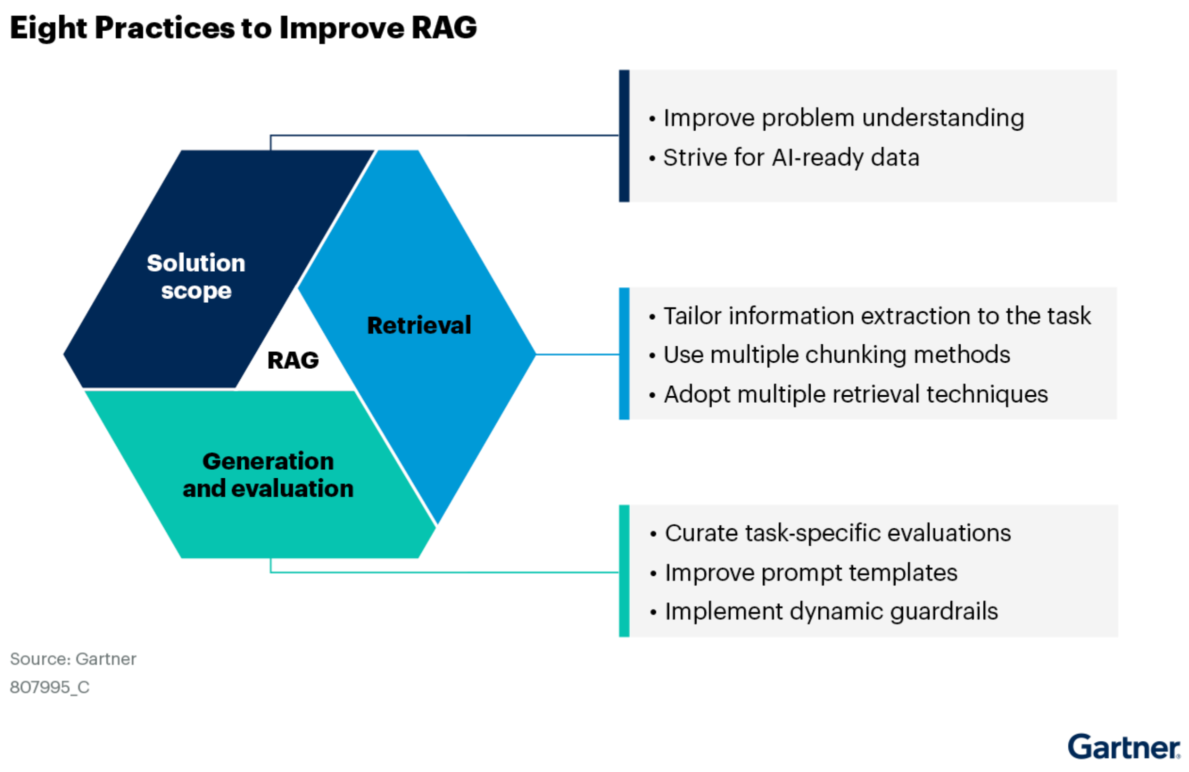
The RAG pattern's ability to combine LLMs with internal context and an easily understood core architecture has made it well known. However, building production-ready RAG-based LLM applications is difficult. Gartner clients have questions, such as:

How do we ensure that the quality and capability of our RAG solution is good enough for production use?

This research will help you improve RAG and advance it from prototype quality to production quality. The guidance in this research covers the following (click links to jump to sections):

- RAG system design principles:
  - Principle 1: Build a specialized RAG system, rather than a general-purpose RAG system
  - Principle 2: Modularize your RAG system with data ingestion, retrieval and generation components
  - Principle 3: Deploy common RAG optimization techniques first
  - Principle 4: Optimize upstream components before you optimize downstream components
  - Principle 5: Collect feedback on RAG system performance and behavior
  - Principle 6: Facilitate collaboration between software engineering, data engineering, and cloud engineering and security teams
- RAG best practices (see Figure 1):
  - Practice 1: Improve problem understanding
  - Practice 2: Strive for AI-ready data
  - Practice 3: Tailor information extraction to the task
  - Practice 4: Use multiple chunking methods
  - Practice 5: Adopt multiple retrieval techniques
  - Practice 6: Curate task-specific evaluations
  - Practice 7: Improve prompt templates
  - Practice 8: Implement dynamic guardrails

Figure 1: Eight Practices to Improve RAG


Eight Practices to Improve RAG
Solution scope
Retrieval
RAG
Generation and evaluation
• Improve problem understanding
• Strive for AI-ready data
• Tailor information extraction to the task
• Use multiple chunking methods
• Adopt multiple retrieval techniques
• Curate task-specific evaluations
• Improve prompt templates
• Implement dynamic guardrails
Source: Gartner
807995_C
Gartner

Note: *This research identifies the key issues that inhibit the quality and production-readiness of RAG-based applications, providing guidance on how to avoid or resolve them. It does* ***not*** *address RAG system cost, application integrations, security, platforms or authentication. It also does not address scaling (user session capacity), high-performance RAG systems or cutting-edge RAG systems. In addition, it does not dive deeply into prompt engineering or evaluation.*

RAG System Design Principles

Back to top

RAG design principles precede practices. Principles are important because they provide overall structure and guidance to organizations building RAG systems. The following curated subset of design principles aligns with common missteps in RAG system design.

Principle 1: Build a Specialized RAG System, Rather Than a General-Purpose RAG System

Back to top

Successful RAG requires you to become more deliberate in what you build. A specialized RAG system has a specific user, use case and data scope, with specific questions aimed at a specific data type. For example, the system may layer Q&A over U.S. Securities and Exchange Commission (SEC) 10-K forms about company similarity or performance.

By contrast, a general-purpose RAG system is designed to provide both granular and general answers for many different types of users. It handles all forms of questions — fact retrieval, comparison, reasoning and programming — on a very broad scope of data (e.g., all the data in your Microsoft SharePoint).

Specialization is necessary because different questions require different types of data preprocessing, retrieval approaches and evaluations. The more varied your solution scope, the harder it is to successfully build all of those components. A more specialized focus enables you to develop prompt templates, evaluations, and data ingestion and retrieval techniques that are tailored to the use case.

Principle 2: Modularize Your RAG System With Data Ingestion, Retrieval and Generation Components

Back to top

Organizations seeking optimization often focus on three components:

- The data ingestion/data preprocessing pipeline to create a knowledge corpus
- The search/retrieval system to fetch the right content for the right use case at the right time
- The generation system, which produces and synthesizes output

You must optimize both the system and its parts to succeed with RAG. For a detailed understanding of the steps involved, see Getting Started With Retrieval-Augmented Generation.

Principle 3: Deploy Common RAG Optimization Techniques First

Back to top

Examples of such techniques include small-to-big retrieval, better information extraction, better chunking, hybrid search, metadata filtering and LLM-based services/processes. These optimizations are currently cheaper and easier to implement than advanced techniques, such as agentic RAG, graph RAG or fine-tuning models.

However, you may not be able to avoid agentic, graph or fine-tuning approaches as you scale up. See the Preparing for Advanced RAG Optimizations section for more information about these advanced techniques.

Principle 4: Optimize Upstream Components Before You Optimize Downstream Components

Back to top

Source content, data ingestion and retrieval systems are very important to RAG. Most RAG systems prioritize context retrieval. Issues during the ingestion and retrieval stages will cause generation failures. One common optimization is to start with the best-quality and most-organized document content as a source corpus for a RAG system, and only later deal with messier data.

Principle 5: Collect Feedback on RAG System Performance and Behavior

Back to top

RAG systems depend on feedback and evaluation mechanisms because of the open-ended nature of response generation. As an AI architect or engineer, you should aim to launch simple RAG applications that you can pilot quickly. During this pilot, where possible, capture prompts from your users, as well as real user feedback. This information is crucial to understanding how your RAG system is performing.

Use this feedback to improve question answering or to mitigate known failure domains. Without a feedback loop, the RAG implementation will be suboptimal at best, especially in terms of LLM output. Also, choose vendors in the AI engineering space that make feedback and telemetry collection easier.

Principle 6: Facilitate Collaboration Between Software Engineering, Data Engineering, and Cloud Engineering and Security Teams

Back to top

Many of the technical challenges of RAG benefit from the knowledge of multiple technical domains. Thinking of a RAG system as an AI product — which can be developed by a product team and supported by platform teams — is a best practice here. (See Organizing Product and Platform Teams to Scale Generative AI Delivery for more information.) For example:

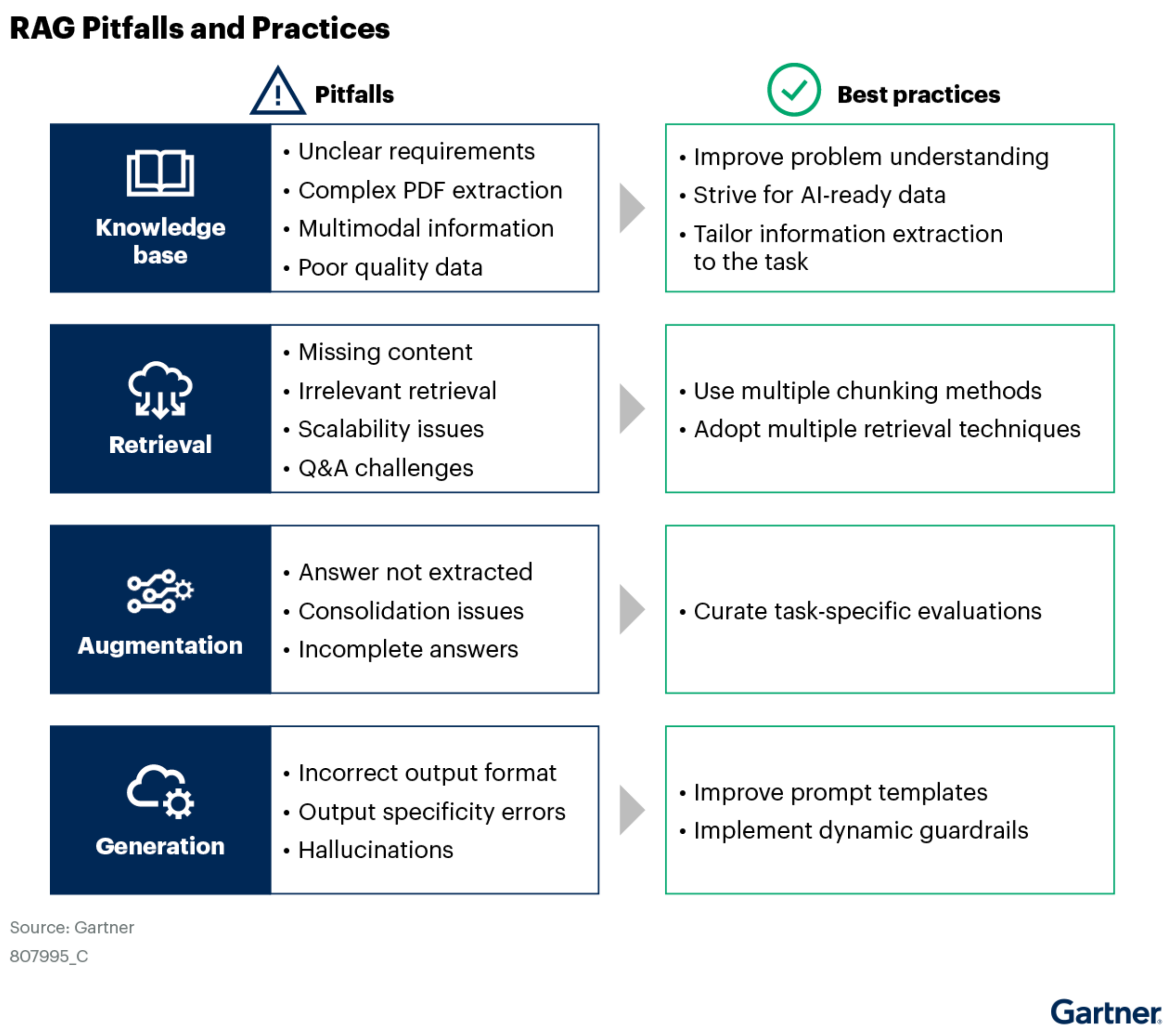
- If your organization has a mature software engineering function, its software engineers will be very good at identifying requirements, working with stakeholders and delivering a working distributed system.
- If your organization has a mature data engineering function, its data engineers will be excellent partners at helping you overcome the data-management-related challenges of RAG. These challenges include working with unstructured context, extracting metadata, creating data pipelines that can update changed documents, and scaling data processing for the RAG system.
- If your organization has a mature cloud engineering practice, its cloud engineers will be able to help you scale the infrastructure and operations processes necessary to deliver the RAG solution.
- Finally, the security team will be crucial to helping you build the right guardrails and processes to secure the RAG-based application.

RAG Best Practices

Back to top

A lot can go wrong on the journey of developing a RAG system. Figure 2 illustrates a subset of the issues you may encounter and maps them to eight best practices for improving a RAG system.

Figure 2: RAG Pitfalls and Practices



The following sections detail each best practice.

Practice 1: Improve Problem Understanding

Back to top

Clarify the RAG Project Requirements Through Discovery

For a software application or a traditional machine learning (ML) use case, user stories are more likely to capture well-defined requirements. In these scenarios, the project scope is well-understood.

However, for a RAG solution, the project scope may relate to a productivity improvement opportunity and corresponding content. The project scope for RAG may exist in the form of future questions that have not been well-defined. Those questions represent user stories, and the RAG development team must discover some of the interactions that they expect users to have with the solution.

The adaptability of chat interfaces, in particular, means opportunities exist for users to try "the unexpected." However, not all RAG systems use chat interfaces or have uniform designs and functions, increasing the need to discover which system your team should build.

In an environment where you must design for the unexpected, one good practice is to reduce uncertainty as much as possible with more discovery.

Address the Foundational Questions for a RAG System

When building an optimal RAG system, you must address three foundational questions:

1. **Discovery:** What is the experience we need the system to provide?
2. **Delivery:** How should we build it?
3. **Go-to-market:** How should we bring it to market? (In other words, how do we go beyond just putting the solution into production? How should we roll it out to users, build awareness and explain how to use it?)

For more information on this process, see Product Owner Essentials.

Technical teams not familiar with established software engineering practices often jump straight into solving technical challenges without adequately addressing these core questions. Understanding the following helps refine the requirements and guide better decision making throughout the development process:

- The specific use case (What experience do we need to provide?)
- The target audience (Who will consume it, and how will we deliver it?)

Starting with the end in mind is crucial. That target audience should also include stakeholders who need to be involved in the discovery phase. Domain experts or business end users will be needed to help determine what requirements are crucial to satisfy for a system, what types of evaluations or tests could be relevant, and what end-to-end experience is desired.

This process will also help you determine whether you need a RAG system at all. If a simple application focused on summarization, translation or content generation will satisfy business user needs, building a RAG solution may not be necessary. In AI, as in software development, it’s important to build the simplest possible technical solution that meets your requirements.

Optimize Your RAG System by Asking Additional Questions

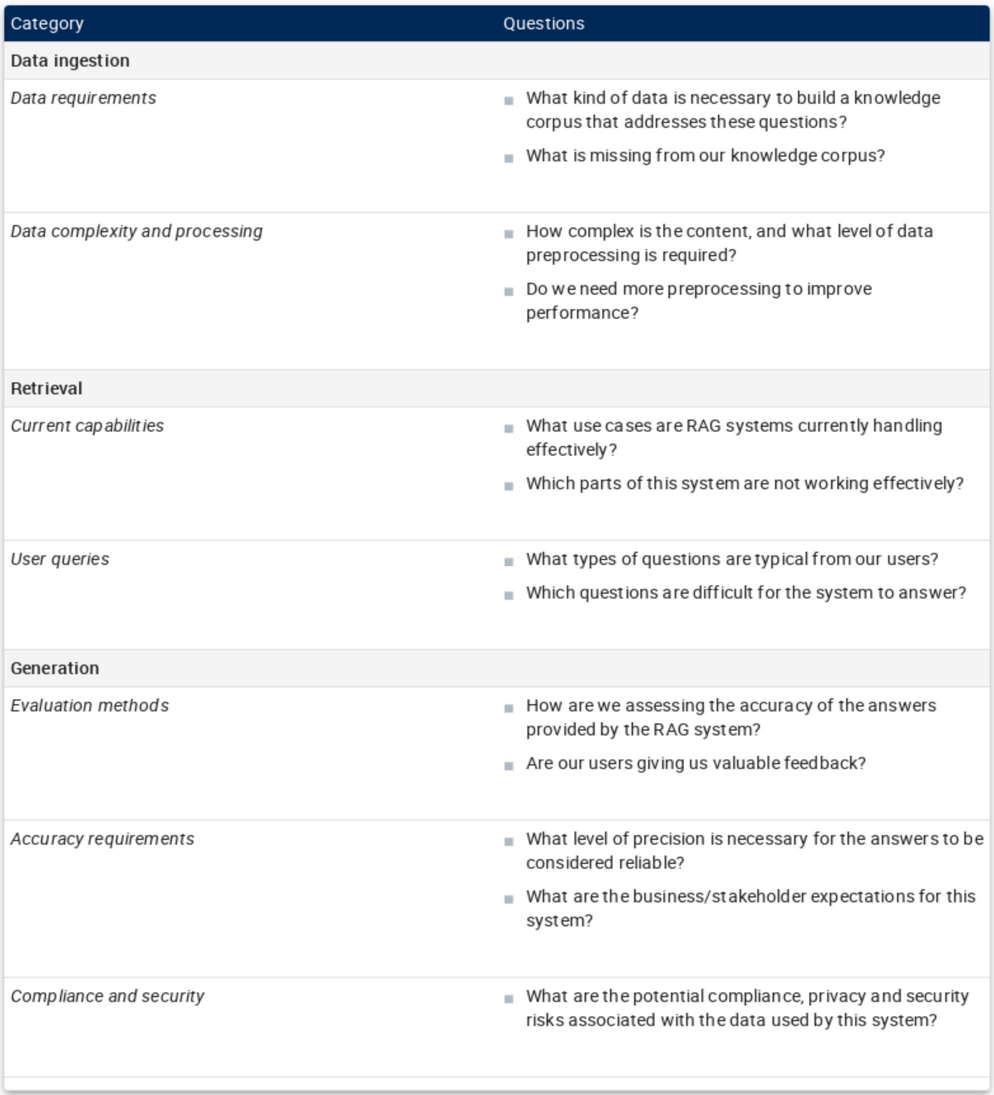
However, you may discover that your challenges relate to the question “How should we build it?” In this case, you can improve the technical aspects of RAG by exploring more low-level questions, such as:

- Do we have the right content for retrieval?
- Is the search working?
- Are we providing the system with the right resources, instructions and evaluations (i.e., prompt templates, observability, guardrails and test prompts)?

These can be accompanied by more generic AI feasibility questions, such as those in Table 1.

Table 1: AI Feasibility Questions

(Enlarged table in Appendix)



Category	Questions
Data ingestion	
Data requirements	▪ What kind of data is necessary to build a knowledge corpus that addresses these questions? ▪ What is missing from our knowledge corpus?
Data complexity and processing	▪ How complex is the content, and what level of data preprocessing is required? ▪ Do we need more preprocessing to improve performance?
Retrieval	
Current capabilities	▪ What use cases are RAG systems currently handling effectively? ▪ Which parts of this system are not working effectively?
User queries	▪ What types of questions are typical from our users? ▪ Which questions are difficult for the system to answer?
Generation	
Evaluation methods	▪ How are we assessing the accuracy of the answers provided by the RAG system? ▪ Are our users giving us valuable feedback?
Accuracy requirements	▪ What level of precision is necessary for the answers to be considered reliable? ▪ What are the business/stakeholder expectations for this system?
Compliance and security	▪ What are the potential compliance, privacy and security risks associated with the data used by this system?

Source: Gartner

The questions are straightforward, but if you answer them without talking to the relevant stakeholders or experts, you are likely to make incomplete or incorrect assumptions. Answers need to be collectively agreed upon by the relevant stakeholders.

Interview Target Customers

Based on Gartner inquiries with clients, AI product teams need to have more story-based conversations with customers to figure out what opportunities, solutions and assumptions exist for the products they are trying to build. Using more open-ended questions to interview the target customers of your AI product can help you better understand their lives/jobs/workflows and where an AI system could assist. The following are great resources for improving your discovery processes:

- Agile Data Science Fundamentals
- Essential Skills for Agile Development
- User Story Essentials

User-centric design, based on detailed requirements gathering, is crucial with RAG systems. Solutions are only successful if they are adopted and if users derive value from their use. A general good practice is to ask internal users:

- What questions they would pose to an AI system in a particular domain
- What answers would be good in that domain (e.g., an HR bot)

Using these questions and answers, you can then, ideally, generate a test dataset. You can also think about the types of documents and data that a RAG system will access, and engage in topic modeling to understand what topics are included in that corpus. From that topic modeling exercise, you can also generate synthetic questions that align with content themes or questions that may not have been explicitly mentioned.

Generally, teams focus more on feasibility than on value, and start with simpler questions that would be more straightforward to answer. Teams also generally prioritize building systems based on higher-quality, lower-sensitivity data sources. This approach minimizes assumptions and focuses on building specialized, useful RAG systems.

Practice 2: Strive for AI-Ready Data

Back to top

Identify Poorly Managed and Poorly Described Source Data

If you consider the principle to address upstream components of RAG before downstream components, one simple optimization for a RAG system is to improve the source data that it ingests. The goal is to reduce:

- **Poorly managed data:** Unstructured data for most organizations has been unknown, risky data. That is, it has been collected and stored without good governance procedures or a clear intent on how to use it. In many cases, content is highly disorganized, to the point where discerning what information it contains is difficult. As described in Developing a Knowledge Management Strategy, these are well-understood issues for search and knowledge management professionals.

- **Poorly described data:** In other scenarios, unstructured data that has not been tagged/evaluated may contain sensitive, outdated, contradictory or incorrect information. This data shouldn't be ingested into a RAG system, but the team building the system won't be aware of the data's sensitivity, inconsistency or inaccuracy.

See Note 1 for a RAG-ready data quality checklist that describes some of the quality dimensions you may be concerned about.

Improve the Quality of Your Unstructured Data

Your organization should focus on ways to improve the quality and associated metadata of the unstructured data that will be used in the RAG system, before that data is ingested into the RAG system. Many organizations have traditionally worked to improve the quality of their structured data. However, they may not have experience improving the quality of unstructured data.

Representative examples of methods to improve unstructured data quality include:

- **Content curation:** In this approach, you apply knowledge management best practices, human effort and data governance capabilities to curate a knowledge base. Developing a Knowledge Management Strategy describes the tasks involved, including tagging, deleting and organizing the data in your knowledge base.
- **Topic modeling:** For messy knowledge bases that are beyond a reasonable effort to remediate, you can use ML techniques, such as latent Dirichlet allocation (LDA), to identify similar topics and group documents into those topics. You can also use vectorization and fine-tuned language models, such as BERTopic. In addition, offerings such as Shelf enable topic modeling and unstructured data management.
- **Content sanitization:** Sanitizing content involves cleaning the corpus of problematic elements, such as sensitive information, overshared information, and inconsistent or outdated information. Mitigate Copilot for Microsoft 365 Risks Through Information Governance provides an example of problems and mitigations in this domain. The sanitization process ensures that the information used by the RAG system not only remains relevant, but also adheres to ethical guidelines and maintains the integrity of the generated content. Regular updates and checks are essential to keep the content current and appropriate.

Practice 3: Tailor Information Extraction to the Task

Back to top

For RAG systems to work, you can't just manage the source content — you have to understand the source content and extract the right information out of it.

Improve Content Extraction

In RAG, data ingestion commonly involves retrieving source documents from a file share, extracting data from those documents and populating them into a well-organized repository. For technical guidance on building these extraction systems, see Working With Semistructured and Unstructured Datasets.

AI architects and engineers building RAG systems must contend with issues related to information extraction. These include content-parsing issues and content interpretation issues.

Mitigate Content-Parsing Issues

Common content-parsing challenges include:

- **Variable text formatting:** Variations in structure and formatting make consistent extraction/chunking of text difficult.
- **Complex tables:** Accurately interpreting headers, nested tables and data types is challenging, and few solutions exist that can handle these tasks consistently well.
- **Difficult-to-parse PDFs:** PDFs are designed for human reading, complicating machine parsing.
- **Complex structured content:** Formats like HTML, XML and JSON require specific parsing techniques for nested structures.
- **Failure to capture information from images, diagrams and charts:** Images, diagrams and charts often contain valuable information, but can be quite difficult to analyze. AI architects and engineers struggle to extract information from visualizations consistently.

- **Document cleanup:** Sometimes, rather than extracting data, you may want to exclude certain information from documents. For example, two components of documents that can be confusing during the retrieval phase are headers and footers. Careful management of document headers and footers is necessary to prevent retrieval results with irrelevant or repetitive information. Deciding whether to remove or append this information can impact the context provided, as headers and footers often contain critical metadata or contextual clues that can aid document identification and categorization.

Some solutions can help with complex content parsing. These include:

- Purpose-built tools, such as Unstructured and Pryon
- Solutions from intelligent document processing (IDP) vendors, such as ABBYY and Hyperscience
- Cloud offerings, such as Microsoft Azure Document Intelligence and Amazon Textract

In addition, open-source options, such as Tabula and PDFminer, are available. LLM Sherpa also offers an interesting approach to PDF parsing.

Mitigate Content Interpretation Issues

Content interpretation challenges, and possible mitigations, include:

- **Jargon:** Many documents contain context-specific information and jargon that can be difficult to interpret without access to additional knowledge. Solutions vary by domain, but in healthcare, products that enable named entity recognition (NER) of health data are useful. John Snow Labs is an example. Often, organizations also look at their own data development platform to label entities, using tools such as Snorkel AI, Scale AI and Datasaur. Open-source NER capabilities, such as GLiNER, can also help.

- **Multimodal interrelated content:** Different content elements in documents (such as headers, color schemes, images and labels) and linked content in other documents are often related. However, these relationships can be challenging to extract/define/understand. Approaches to help interpret multimodal content include:
    - Solutions, such as Pryon and Shelf
    - RAG-in-a box capabilities, such as Glean, Ask Sage and Palantir

Extract the Right Metadata

Effective metadata capture involves identifying and storing relevant information, such as document type, author, last-updated date and page numbers. In a structured-data context, this metadata might include table descriptions, schema or generated questions that the table data can answer. You need to systematically tag and integrate this metadata into the RAG system (typically, the vector database) to enable sophisticated filtering and more targeted retrieval, thereby improving the relevance and precision of the search results.

“Metadata filtering” is the process of extracting metadata, associating it with your embedded content chunks and using it to filter results. Vector databases and search services allow you to associate metadata, which you can then filter over.

In addition to structured information, some other types of metadata you might want to capture include:

- Relationships within and between documents/concepts (i.e., NER and entity linking in documents, or linked documents such as related policies)
- Information from tables or graphics (e.g., cell, chart, diagram, graphic or image descriptions)

Capture Relationships Within and Between Documents/Concepts

Capturing these relationships involves understanding both:

- **The internal structure of a document:** Identifying sections/divisions in a document enhances the structural understanding of that document. This task involves recognizing and tagging various document segments, such as headers, chapters or subsections. Open-source projects and document AI tools can help extract this data. Examples include LayoutPDFReader, Azure AI Document Intelligence or Amazon Textract. The extracted data can then enable intelligent chunking during the text-split phase. Improved structural recognition supports more effective content-aware chunking strategies that align with the document's inherent organization, leading to better retrieval outcomes.
- The document's connections to other documents: Analyzing how different documents or sections are interrelated can be crucial for tasks such as retrieving citations, cross-referencing and building comprehensive knowledge bases. To map relationships, you can use flat tables for tagging snippets (e.g., tagging headers, or hyperlinks to other documents). For more advanced relationship mapping, GraphRAG is a promising approach, as described later in the Preparing for Advanced RAG Optimizations section.

Extract Information From Tables/Graphics

This conversion involves extracting readable text from tables and interpreting graphical data, such as charts or diagrams, into descriptive text that can be processed by the system. Skilled AI engineers and architects use multimodal LLMs to generate text descriptions of visual elements. By storing these connected entities with linked metadata, they enable the RAG system to understand and utilize all available information within the documents more comprehensively.

Table extraction is a common and important information extraction task, with a fair number of available tools, such as Unstructured and LlamaParse. Extracting structured information from charts and diagrams, however, is much more challenging. Although possible, it requires a very task-specific, extremely customized model (e.g., a Plot2Txt converter model for each primary plot type you have in your organization).

Improve Your Embedding Approach

Embedding models are necessary to create vector embeddings. Vector embeddings provide a way to convert textual information into numerical data. These vector embeddings allow RAG systems to efficiently retrieve and utilize relevant information. Vector databases enable the core function of “similarity search,” which is crucial to many RAG implementations. Consequently, it is important to improve your approach to generating embeddings.

However, not every RAG system uses vector embeddings and vector databases for retrieval. Traditional approaches to indexing, including lexical ranking systems such as BM25, are also available. These are separate from embedding and semantic search approaches. Moreover, some advanced use cases leverage LLMs to interpret questions and/or generate graph queries to answer certain types of RAG system questions.

Review/Update Your Embeddings

Regularly updating the embeddings to reflect the latest data ensures that the system remains effective as the underlying data evolves. But for more accessible embedding optimization, consider visualizing and reviewing what embeddings get clustered and which relationships and differences are being captured.

Open-source tools, such as Feder or Parallax, can help with visualizing embeddings. AI engineers with data science skill sets will also often use dimensionality reduction techniques to visualize specific embedding dimensions. Examples of these include t-distributed stochastic neighbor embedding (t-SNE), principal component analysis (PCA) or uniform manifold approximation and projection (UMAP).

Adopt Specialized Embeddings

Alternatively, experimenting with different types of embeddings, such as graph embeddings for interconnected data or transformer-based embeddings for deeper semantic understanding, can offer substantial improvements. Consider adopting specialized approaches to embedding models for specific high-difficulty use cases. These use cases may include intricate domains, such as genetics or legal LLMs. An example of a specialized embedding model is phospho’s Intent Embed, which attempts to embed a user intent with more clarity compared with generic embeddings.

Moreover, you can experiment by adjusting the number of dimensions that embedding models generate. These default dimensions might be smaller (e.g., 768, 1024) or larger (e.g., 1536, 3072). More or fewer embedding dimensions can provide performance or context extraction benefits.

Consider Fine-Tuning Your Embeddings

Fine-tuning the embedding model is an optimization strategy that organizations sometimes pursue. It might involve fine-tuning existing embeddings with domain-specific data to capture unique terminologies and concepts better. The embedding model space has become more diverse. Therefore, as a starting point, evaluating and selecting from a broader set of embedding models, such as those available via the Massive Text Embedding Benchmark (MTEB), is a strong option.

Practice 4: Use Multiple Chunking Methods

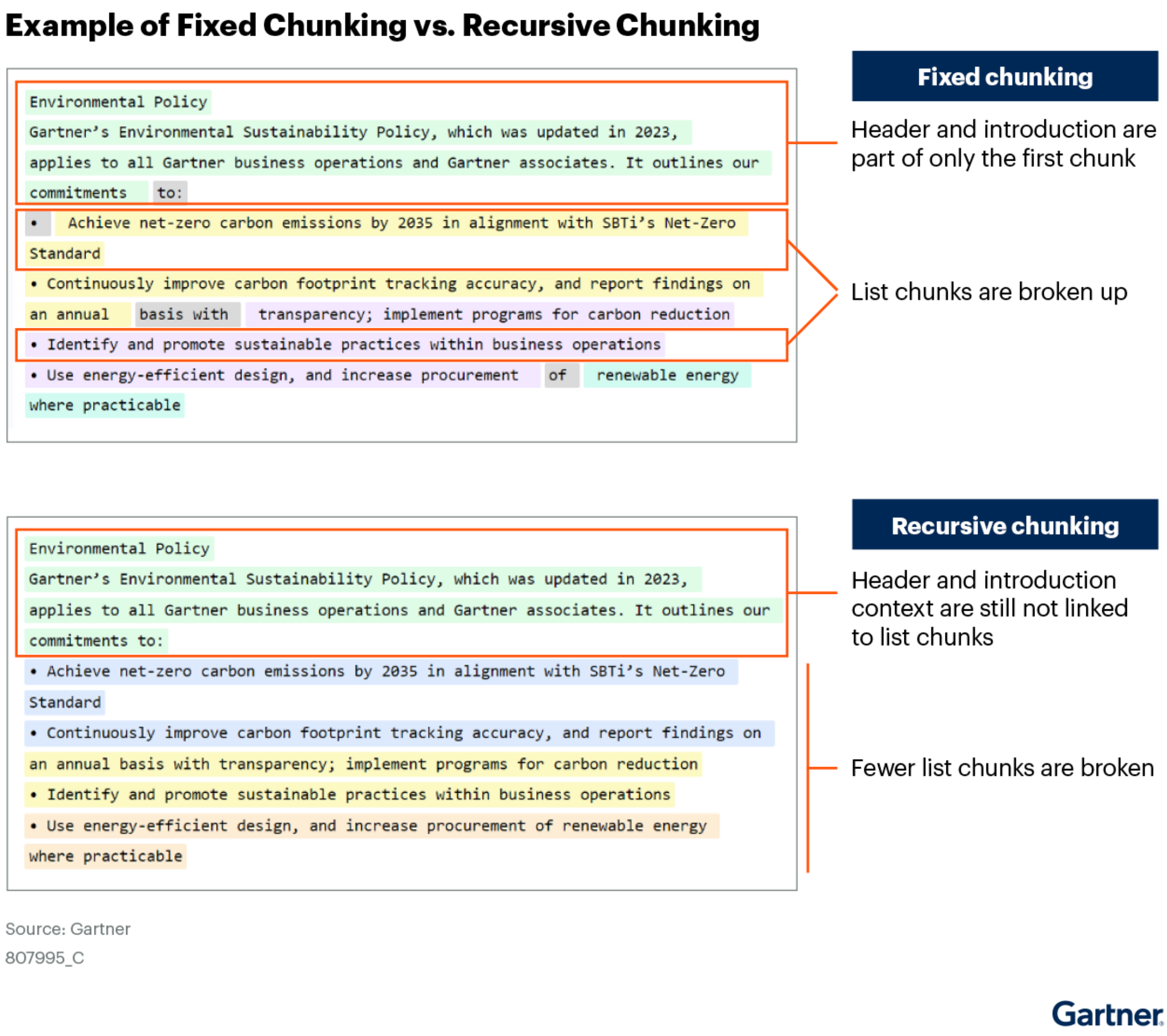
Back to top

Chunking involves breaking down large texts or datasets into smaller, more manageable and semantically meaningful units. Effective chunking ensures that the content retrieved from a vector database is both relevant and semantically coherent. This outcome is crucial for embedding content effectively and for ensuring the performance of LLM RAG applications, where the quality of the response depends heavily on the precision of the underlying data retrieval.

Chunking approaches are important to RAG systems because they have to fit information into a context window. Thus, how you chunk information will impact the performance, completeness and relevance of your retrievals. However, multiple chunking techniques exist. Moreover, the ideal chunking strategy also depends on the task and the question asked. Depending on the content and the task, different chunking techniques will be necessary, and you will often need to deploy more than one chunking approach in your RAG system.

Consider an example from Gartner's 2023 Corporate Responsibility Report (see Figure 3). If you apply a fixed chunking approach to the section on environment policy, you are likely to encounter some challenges. List chunks get broken up, and the header and introduction contain information that might be worth appending in other chunks, depending on the task.

Figure 3: Example of Fixed Chunking vs. Recursive Chunking


Example of Fixed Chunking vs. Recursive Chunking
Environmental Policy
Gartner's Environmental Sustainability Policy, which was updated in 2023,
applies to all Gartner business operations and Gartner associates. It outlines our
commitments to:
• Achieve net-zero carbon emissions by 2035 in alignment with SBTi's Net-Zero Standard
• Continuously improve carbon footprint tracking accuracy, and report findings on an annual basis with transparency; implement programs for carbon reduction
• Identify and promote sustainable practices within business operations
• Use energy-efficient design, and increase procurement of renewable energy where practicable
Fixed chunking
Header and introduction are part of only the first chunk
List chunks are broken up
Recursive chunking
Header and introduction context are still not linked to list chunks
Fewer list chunks are broken
Gartner

Source: Gartner
807995_C

The second approach, recursing chunking, utilizes the LangChain RecursiveCharacterTextSplitter for chunking. As text is split, it accounts for several types of delimiters and applies a recursive set of techniques to rightsize the chunk. This approach preserves the list items with less fragmentation. However, it's just one example of an improvement to chunking, specifically for list data. More complex techniques are needed for other questions or data types.

Different Chunking Strategies

Several chunking strategies can be tailored to meet different application needs:

- **Fixed-size chunking:** This straightforward approach involves dividing text into uniformly sized chunks. A fixed-size chunk is computationally efficient and still effective in many cases. However, because fixed chunking does not respect sentences or other content groupings, it's likely to create fragmented chunks. Generally, fixed-size chunking is inferior to more advanced methods. Despite this fact, it is often the default approach for many RAG solutions.

- **Recursive chunking:** This strategy applies different criteria iteratively to refine the chunk sizes and boundaries. This approach introduces additional intelligence to keep all paragraphs (and then sentences, and then words) together as long as possible. Recursive chunking is useful when you want to chunk more efficiently but don't have access to content-aware chunking software or capabilities.

- **Content-aware chunking:** This method permits chunks that align more closely with breaks in the text or other structural elements, enhancing semantic coherence. In a simple example, this approach might involve using a tool like Unstructured or Amazon Textract to analyze document structure and chunk according to the layout of the document. For instance, chunking a table could ideally store the table as a separate chunk with the hierarchy preserved. Alternatively, a long table could be chunked into separate sections, but with the corresponding title and descriptive information in each chunk. The type of content will affect the chunking. For instance, when you are chunking an HTML page, it is typically useful to capture the tag pairs that encapsulate sections (e.g., *<h2> text </h2>)*. Keeping these pairs together is important for semantic meaning. Vendors and tools vary in their support for content-aware chunking. However, we expect more capabilities in this domain to be released, as organizations require better RAG retrieval performance.

- **Semantic chunking:** This is a more sophisticated approach that uses embeddings to dynamically adjust chunk sizes based on themes and context continuity. This approach is ideal for complex documents. However, because of the use of embeddings, this approach will involve more computation and more complexity. Some tools, such as Unstructured, enable this capability, and some vector databases will endorse this approach to increase the usage of their functions. The need for this approach will depend on how complex and diverse the questions and corpus being used for chunking are.

- **Hierarchical chunking:** Different approaches to nested or hierarchical structures can improve chunking. A common approach with chunking and metadata is to combine individual chunks with global document metadata. However, global metadata doesn't always work because documents often have sections, thus requiring relevant context at a per-section level. In one example of a hierarchical chunking implementation, smaller document chunks are enriched with subdocument summaries/metadata. In another example of hierarchical chunking within the small-to-big retrieval style, parent chunks are defined and then linked to smaller, more specific chunks. You can, for example, search over smaller chunks (which can be summaries) and then insert the parent chunk into the response (which would be a larger, richer contextual response). Different tools allow you to build hierarchical chunking. Examples include LlamaIndex, which has Llama Packs, and AWS, which offers hierarchical chunking inside Amazon Bedrock knowledge bases.

Factors Influencing Chunk Size Determination

AI architects and engineers need to weigh several factors when determining chunk sizes:

- **Content type and length:** Extensive documents (e.g., research articles) and brief texts (e.g., social media posts) may require different chunk sizes. Larger chunks with more metadata may be necessary for longer documents to preserve complete bits of context.
- **Query complexity:** For complex queries, larger chunks with richer information extraction might be required to capture all relevant contexts and nuances within the text. Examples of such queries include reasoning questions that require a multihop retrieval, or strategy questions that cover a lot of domain topics in an organization. Sometimes, the necessary data is in more than one document, making other elements of your data ingestion/preprocessing strategy, such as metadata association and document linkages, crucial.
- **Application requirements:** The specific needs of the application also influence chunking decisions. Such factors may include the constraints of downstream language models (e.g., token limits) and the goals of the application (e.g., detailed semantic search or quick conversational responses).

The Impact of Large Context Windows

The latest LLMs feature extensive context windows, offering new possibilities for organizations developing RAG systems. AI architects and engineers can now input substantial amounts of content into an LLM's text prompt. This capability is particularly useful in pilot scenarios to evaluate how well a model retrieves information from specific content types, such as videos or books.

However, chunking-based approaches remain crucial as organizations scale. Smaller context is often more cost-efficient, more performant, less risky and easier to scale. (See Note 2 for further information.)

How to Approach Chunking

1. **Employ multiple chunking techniques:**

   - Utilize multiple chunking methods, tailored to different document types (e.g., text, video transcripts and technical manuals), simultaneously. This diversification ensures that each document is processed in the most efficient manner to enhance retrieval accuracy.

   - Be mindful of the trade-off between the number of embedding calls and the system's performance. Excessive embedding calls can increase both cost and latency. Optimize by balancing the granularity of chunks with the frequency of embedding operations.

2. **Leverage larger context models judiciously:**

   - Larger models with extended context can simplify chunking by reducing the need for extensive fragmentation of documents. This approach can streamline processing and improve coherence in information retrieval.

   - However, while larger context models can alleviate some chunking complexities, they introduce new trade-offs, such as higher computational costs and potential performance bottlenecks. Assess these factors carefully to ensure that the benefits outweigh the drawbacks in your specific use case.

Practice 5: Adopt Multiple Retrieval Techniques

Back to top

Building an effective search capability is difficult. It requires many complex trade-offs to account for:

- The vast amount of information to search
- The many different approaches to search
- The idiosyncrasies of users, who don't always know what they want or how to ask for it

These problems get amplified when you add the complexity of a RAG system that is expected to query, retrieve and synthesize the right search results on behalf of an often-imperfect user prompt. Moreover, the questions you ask affect how the RAG system will need to retrieve information to answer them.

Q&A Challenges

Designing a RAG system to satisfy fact-retrieval questions is often easier than designing one to satisfy reasoning or comparison questions, which may require information that is in other places or in many different sources. Other questions, such as complex analytics questions, require sophisticated queries against structured data. Although many teams would like to deploy natural language text-to-SQL capabilities using RAG, these types of questions often generate flawed outputs.

AI architects and engineers puzzle over these issues. Because RAG systems have multiple components, debugging answers is complex. AI engineers and architects must evaluate whether the problem is the fault of the component, the model, the prompt, the retrieval, the ingestion or something else.

From the perspective of search, several foundational issues commonly pop up in RAG systems:

- **Missing content:** When crucial parts of content are missing during retrieval, the system may provide incomplete or fragmented answers, reducing the quality/reliability of the RAG system.

- **Irrelevant retrievals:** Similarly, retrieval of irrelevant document chunks introduces the possibility of off-topic answers. Moreover, filtering through irrelevant data increases the overhead (and cost) of the RAG system and dilutes the quality of generated responses.

- **Struggles to optimize retrieval relevance with the right cost structure:** AI engineers and architects must build retrieval systems that provide relevant context but with sustainable costs. Overly costly or complex retrieval approaches may not scale, but overly simplistic retrievals may not perform well.

Scaling Challenges

A RAG system that works well with hundreds of documents may not work as efficiently at a much larger scale. Ingesting a few PDFs is straightforward, but handling billions of records introduces significant infrastructure and engineering challenges. Moreover, parallelization of requests, retry mechanisms and appropriate distributed systems are critical.

Teams are often ill-equipped to understand data volume, ingestion time requirements, search latency and costs for RAG systems. For example, generating embeddings becomes challenging with large datasets due to rate limits, retry logic and the need for self-hosted models. Efficient mechanisms are needed to detect and reembed only the necessary data to manage costs effectively.

Retrieval Optimizations

Anecdotally, the most common retrieval optimizations are:

- Separating data strategies for retrieval and response (i.e., small-to-big retrieval)
- Filtering based on metadata
- Using hybrid search
- Implementing preretrieval and postretrieval optimizations, such as query transformation techniques and reranking techniques

Separating Your Data Strategies for Retrieval and Response (i.e., Small-to-Big Retrieval)

Optimizing RAG systems for retrieval and response first involves acknowledging that these two systems require separate optimizations.

Consequently, organizations are decoupling the data strategies for retrieval and response generation. This approach enables specialized data handling that enhances both the retrieval accuracy and the relevance of the generated content.

For retrieval, using concise and highly indexed chunks (such as document summaries or sentences) ensures efficient and precise searches. For response generation, richer, more detailed data chunks provide the necessary context for generating accurate and informative responses. Examples include longer text chunks or sentence chunks that include windowing to add both the previous chunk and the subsequent chunk.

The goal with small-to-big is to solve for:

- Retrieval relevance with better retrieval-optimized chunking
- Missing content with more content inclusion during response

Filtering Based on Metadata

Metadata filtering is a powerful technique to optimize information retrieval. In a RAG system that incorporates metadata filtering, users (or a specialized LLM) select upfront metadata filters (e.g., options to filter by publication date, author or topic). These filters allow for search over a more specific set of relevant documents. This targeted retrieval is particularly effective in environments with large, diverse datasets, where precision is critical. For example, in a medical research RAG system, filtering by recent publication dates or specific medical conditions can lead to more accurate and clinically relevant information retrieval.

The metadata that is useful for filtering is also useful for managing unstructured data. For example, once date-related metadata is captured, it could also enable data beyond a certain age to be purged from the index or the vector database.

Moreover, filtering helps with inference speed, as it lessens the need to perform semantic search on documents that are not relevant. In essence, metadata filtering enables a hierarchy of search. The system filters based on metadata and then performs a semantic search on that subset.

Using Hybrid Search

Hybrid search strategies most commonly combine lexical search with semantic search, offering a comprehensive solution to the challenges of retrieval in RAG systems. Lexical search quickly locates documents containing specific keywords, while semantic search, powered by embeddings, interprets the context and nuances of the query. This combination ensures that the retrieval process remains robust against varied query formulations and can handle synonyms, related terms and nuanced queries effectively.

Hybrid search is often combined with metadata filtering to enable efficient, performant retrieval.

Implementing Preretrieval and Postretrieval Optimizations

Many types of preretrieval and postretrieval optimization exist. Common preretrieval optimizations include refining the query (rewriting) or applying the hybrid search approach described above. Postretrieval optimizations could involve reranking, using either:

- Specialized models, such as cross-encoders
- Composite reranking algorithms that consider user feedback, recent interactions and other dynamic factors to adjust the relevance of retrieved documents continually

Additionally, implementing ML models to learn from past retrievals can dynamically improve the retrieval process over time.

Preretrieval and postretrieval optimizations are important to retrieval quality. Therefore, it is worth considering popular methods in this domain. Below, we discuss query transformation techniques and reranking techniques.

Query Transformation Techniques

These techniques modify the input prompt to align better with the structure and expectations of the knowledge base and language model. Such techniques include:

- **Expanding or rephrasing:** Adjusting the input to capture additional context or alternative formulations
- **Entity and keyword extraction:** Identifying key entities, concepts or keywords to focus the retrieval process

- **Translation:** Converting the input to a different language or domain-specific terminology
- **Augmentation:** Adding extra information, such as user preferences or task-specific constraints

Reranking Techniques

These techniques enhance the ranking of retrieved results to ensure the most relevant information is used in the generation process. Such techniques include:

- **Additional signals and features:** Using factors like semantic similarity, source credibility and task-specific relevance to reevaluate and reorder results
- **ML models:** Applying neural ranking models or reinforcement-learning-based approaches to infer optimal ranking criteria from data
- **User feedback:** Incorporating user feedback or interaction data to fine-tune the ranking algorithm and better align with user preferences

Combining multiple approaches in this domain is known as “composite techniques” or “RAG fusion.” RAG fusion utilizes multiple generated queries and reciprocal rank fusion (RRF) techniques to unify the generated queries and ranked responses into a better output.

Scaling Optimizations

An important take-away for improving your RAG system is that the vector database’s performance and scalability are sometimes crucial. In those scenarios, you should seriously pursue the best vector database for your compute, processing and scaling requirements. Other times, use cases will require vector databases with feature richness, such as native capabilities or integrations for metadata filtering, hybrid search, rewriting and agentic workflows.

Rather than optimizing for performance, optimizing for quality is more important in some vector database use cases. In an emerging trend, organizations are also reverting to lexical search for some use cases. They are finding that users and their use case can scale and perform adequately when lexical search is combined with data preprocessing and advanced preretrieval and postretrieval optimizations. Don’t be afraid to adopt simpler, proven search techniques, or to combine them.

The following are some system design suggestions for building more scalable RAG systems:

- **Maximize parallelization:** Maximize parallelization to meet ingestion time requirements and enable efficient data reading.
- **Optimize chunking and metadata extraction:** Carefully chunk data based on source type, extract relevant metadata fields and clean anomalies before embedding.
- **Apply conditional and parallel embedding:** Perform embedding only when necessary and parallelize it according to system constraints and external API limits.
- **Optimize vector database storage:** Understand the potential pitfalls and take steps to mitigate them:
  - *Resource management:* Optimize compute resources, determine whether to adopt a database that is self-hosted or managed, and improve the observability and monitoring systems that can capture system telemetry.
  - *Performance factors:* Understand and address issues related to data sharding, latency, compression and inefficiencies in algorithms like hierarchical navigable small world (HNSW) for storing identical vectors.
  - *Potential bottlenecks:* Optimize the ingestion process into the vector database so that it does not become a system bottleneck.

For a useful example of how one team improved and scaled its RAG system, see this case study from Neum AI (via Medium).

Practice 6: Curate Task-Specific Evaluation

Back to top

Implementing evaluation practices is essential for identifying how well your RAG system is performing componentwise and overall. It also helps improve response times and maintain the integrity of your system. The problem is that many ML methods of evaluation don’t work that well in RAG. In addition, RAG presents a plethora of variables you may want to evaluate.

Identify Evaluation Areas to Focus On

In addition to the previously cited RAG issues, a number of challenges arise in the RAG generation phase that you may need to evaluate and address:

- **Answer not extracted:** Sometimes, the RAG system doesn't successfully extract the data required to answer the question.
- **Consolidation issues:** Most RAG systems require some form of synthesis. If a very large amount of context is passed, synthesizing the right answer from that context can be challenging. Multiple retrieved sources could have conflicting information, leading to lower-quality responses.
- **Incomplete answers:** Retrieving only part of an answer can affect system accuracy. These answers often stem from retrieved content that lacks critical details or is missing the broader context necessary to address the question completely.
- **Incorrect output format:** Users often expect answers to be presented in a specific format, such as a table. If the output format is incorrect (e.g., unordered text), they may be confused or disappointed. Other common challenges with output format include prompts in one language that return results in a different language, or responses that don't have the right tone.
- **Output specificity errors:** Specificity errors can also plague RAG systems. For example, overly general or overly specific answers may not meet certain users' expectations. User data flywheels (discussed below) or LLM-based evaluation techniques can help refine specificity.
- **Hallucinations/incorrect answers:** Hallucinations, which are typically defined as incorrect or unverifiable responses, are also a big issue for RAG systems. Although certain optimizations can reduce hallucinations, most RAG implementers have not been able to completely eliminate them. Experienced LLM developers still see hallucinations 2% to 10% of the time (or more)!

This list is just a sample of the areas you may want to evaluate. As described in Key Skills to Build LLMs With Retrieval-Augmented Generation, organizations should evaluate and observe:

- RAG output quality
- Application performance
- Security
- Data privacy
- Cost

- Regulatory compliance

Consequently, what may start as a simple set of test prompts and evaluations can balloon into a full evaluation and observability system, with a broad and diverse range of test prompts and observability capabilities. To learn about frameworks for AI observability and generative AI (GenAI) security, see:

- Introduce AI Observability to Supervise Generative AI
- Generative AI Adoption: Top Security Threats, Risks and Mitigations
- Quick Answer: Implementing Data Access Controls for Generative AI Assistants

Understand When You Need to Evaluate LLMs and RAG Systems, and What Methods to Use

RAG system design has multiple types and phases of evaluation:

- **Model/component evaluation:** During the build and design of the RAG system, AI engineers and architects must evaluate generation models, other RAG system components (e.g., the embedding model) and the vector database for their individual capability. How to Choose the Right Large Language Model, a Gartner webinar, provides a view into this process for the LLM component choice.
- **RAG development evaluation:** This phase includes:
  - *Prototype evaluation:* Evaluating the working RAG prototype
  - *Preproduction evaluation:* Evaluating the RAG system during the pilot phase and testing it before it goes into production
  - *Production evaluation:* Evaluating the RAG system in production

To enhance retrievals in RAG systems, you must select robust evaluation methods that reflect the real-world effectiveness of the system. However, considering the complexity and varied applications of RAG, you also need to incorporate domain-specific metrics that capture the nuances of different use cases.

For instance, in a legal document retrieval system, accuracy might be measured by the system's ability to retrieve all relevant case laws without omitting any critical references. See Note 3 for sample metrics that you can use to evaluate RAG system quality. The specific RAG implementation teams will determine the broader mix of evaluations.

Many organizations will start with expert-, tool- and user-focused evaluation methods. They will employ experts to score accuracy and other qualities of generated responses. They may also employ LLM evaluation and monitoring products, such as Arize AI, Galileo, TrojAI or CalypsoAI. A more extensive list of vendors is available in Innovation Guide for Generative AI in Trust, Risk and Security Management.

Implement User Data Flywheels

Production AI systems tend to be open-ended and nondeterministic. Therefore, you need user feedback to validate whether the system is performing in a helpful way. In RAG systems, this feedback is quite crucial. It can not only provide a very valuable signal as to how the system is performing, but also feed the creation of a dataset that can be used for further optimization, testing or fine-tuning.

However, it is challenging to build user data flywheels for RAG systems. We recommend learning from other domains, such as product analytics, and implementing flywheels that:

- Offer UX features for feedback (e.g., thumbs up or thumbs down)
- Prompt for feedback (i.e., ask for additional detail)
- Collect implicit feedback (e.g., observe user interactions with prompts)

As noted above, a common way to collect implicit feedback is to observe the user's interactions with prompts. If a user rewrites the same prompt a few times, it could indicate that the user doesn't like the generated response.

For examples of how software can be designed to collect feedback for AI systems, review how Github Copilot and Midjourney have implemented product features that provide a range of positive and negative signals.

Review Context and Outputs Manually

As advanced as LLMs have become, many teams still rely on manual review to get a feel for the way that a RAG system is functioning. Although it is difficult to scale, manual review is crucial to ensuring the quality and relevance of RAG inputs and outputs. AI engineers and architects should periodically examine the selected context and outputs to identify any inaccuracies, irrelevant information or biases that might have been introduced during the data preprocessing, data ingestion, retrieval and generation stages.

Manual review helps you get a quick understanding of the state of your RAG system and develop further areas of evaluation.

Train LLM Evaluators

Evaluation as a manual task does not scale. Consequently, organizations that are advanced in RAG implementation will be tempted to purchase or train specialized LLM-based systems for evaluation. Specialized LLM evaluators are not easy to develop, but they are worth mentioning as a method of improving and scaling evaluation. Eugene Yan's blog provides two guides [1,2] that you may find particularly useful.

Practice 7: Improve Prompt Templates

Back to top

To get the most out of a RAG system, you have to figure out the right instructions to provide. Instructions are often offered via a prompt template. Prompt templates are predefined recipes for generating prompts for language models. A template may include instructions, few-shot examples and specific context. AI architects and engineers may find that they have to develop multiple prompt templates for different use cases, LLMs or questions. This section provides some tips for improving your prompt template.

Employ Prompt Engineering

Prompt engineering is a new discipline. It is the craft of designing and optimizing user requests to an LLM or LLM-based chatbot in order to get the most effective result. Based on Gartner inquiries with clients, AI engineers and architects are finding that prompting LLMs to generate their desired outputs in a consistent, versioned and adaptable manner is challenging. In the context of developing a RAG system, prompt engineering typically involves building out the prompt template and instructions to guide the right generation.

How to Engineer Effective Prompts for Large Language Models provides a useful overview of the best practices to decompose prompts into subcomponents and to optimize them. One method worth calling out is chain-of-thought (CoT) prompting.

CoT Prompting

CoT prompting encourages the language model to articulate its reasoning process, step by step, before arriving at a conclusion. When properly implemented, this technique can not only enhance the transparency of the model's decision-making process, but also reduce the occurrence of hallucinations.

A common approach is to design prompts that guide the model through a logical sequence of thoughts. For instance, in a financial analysis task, the user could prompt the model to first identify key financial metrics, then analyze trends, and finally synthesize these insights into a forecast. Alternatively, an LLM could generate a series of these prompts based on the user prompt, as LLMs are also effective prompt engineers.

In the RAG domain, CoT prompting can help avoid consolidation issues, incomplete answers, incorrect answers and output specificity errors.

Stop Passing Disorganized Context

Relying solely on a disorganized approach for context representation can lead to suboptimal outcomes. AI engineers and architects must put some thought into the prompt template and the way that context is organized. Unordered collections of context/chunks will potentially confuse the model or lack valuable syntactic and semantic information. Instead, transitioning to one of the following can improve performance:

- More sophisticated context representations, with better CoT prompting built into the prompt template
- Structures that maintain the order and relationships between the different bits of context that are being passed

Thinking about how much context is essential is important as well. Many RAG systems fail because too much context is passed (not just too little), creating "lost in the middle" problems for LLMs. Compression of context can help mitigate this issue.

Compression of Context

Compressing context involves condensing information without losing critical content, allowing the model to handle more data within the limited context window of current LLMs. Compression is often essential for scenarios such as complex queries, extensive background information, or very dense, complicated reference material. Different techniques can enable this outcome:

- Automatic summarization techniques reduce the length of the documents while preserving key information.
- Advanced tokenization strategies reduce the number of tokens used to represent the data.

- More efficient embedding techniques can capture more information in fewer tokens. For example, you can direct the parser to ignore less meaningful parts of speech, such as adverbs and prepositions.
- Retrieval algorithms can be enhanced to prioritize parts of the context that are most likely to be relevant to the query.

Practice 8: Implement Dynamic Guardrails

Back to top

Implementing guardrails involves setting up system checks that ensure the outputs adhere to expectations. These expectations include adhering to factual accuracy, conforming to expected structural formats, and avoiding other types of risks, such as sensitive data leakage. Guardrails are particularly crucial in RAG applications that require data extraction, where outputs must be structured for seamless integration into databases or downstream applications.

RAG Guardrails

Guardrails for RAG systems are relatively immature, and will require a fair amount of custom development and integration by AI engineers and architects. AI engineers and architects will typically define clear templates or schemas that the RAG system should follow when generating outputs. For example, in a legal document extraction system, outputs could be structured to categorize information into predefined legal concepts.

Implementing guardrails requires a lot of task-specific optimization. For example, guardrails can be:

- Prompt-based (e.g., NVIDIA NeMo)
- Rule-based (leveraging metadata)
- Based on explicit discriminative AI (binary classifiers)
- Small language model (SLM)-based (e.g., Meta Llama Guard and Guardrails AI)

For more information, see Use Model Guardrails to Regulate Generative AI Output and Behavior.

Observability

Observability is crucial here as well. AI engineers and architects often integrate validation layers that automatically check the conformity of the generated outputs to the defined templates.

Use ML classifiers or rule-based systems to flag and correct deviations. Innovation Guide for Generative AI in Trust, Risk and Security Management provides a nice overview of observability tools in this domain.

Defensive UX

Defensive UX involves designing the user interface to anticipate and mitigate potential issues by providing additional information or prompting users for clarification or details when needed. This approach helps refine the input to the RAG system, leading to more accurate and relevant outputs. Defensive UX can include:

- **Clarification prompts:** When a user query is ambiguous or lacks sufficient detail (as detected by a specialized guardrail, for example), the system automatically prompts the user to provide more specific information or to choose from a set of options that clarify intent.
- **User input validation:** A system that offers real-time validation of user inputs to catch and correct errors or inconsistencies before they are processed by the RAG system is another useful guardrail that also has a defensive UX benefit. The real-time validation could include spell-check, format validation, logical consistency checks, or removal of certain irrelevant content from prompts.
- **Adaptive interfaces:** You should design user interfaces that adapt based on the user's interaction history and the complexity of the task. For complex tasks, guide the user through a step-by-step input process that ensures all necessary information is collected.

Be very careful about generalizing user feedback, because it is often sparse and difficult to generalize without a broader understanding of the user workflow. For this reason, you should also engage in user discovery and interviews to learn more, via story-based questions, about how users work and employ AI systems like this.

Preparing for Advanced RAG Optimizations

Some organizations have been adopting agentic RAG, graph RAG or fine-tuning approaches to improve their RAG systems:

- **Agentic RAG systems:** AI agents are autonomous or semiautonomous software entities that use AI techniques to perceive, make decisions, take actions and achieve goals in their digital or physical environments. In the context of RAG, AI agents are developed to make decisions, take actions and achieve goals related to successful retrieval of relevant context and generation of a desired response. These agents could be specialized in certain types of retrievals, or they could evaluate retrievals and execute additional actions as necessary.
  - *How to prepare:* A common theme of this research is that no single approach will be optimal for every scenario. For that reason, agentic systems, which can identify different scenarios and different actions to take, will be a powerful improvement over existing methods. See Innovation Insight: AI Agents for insight on how these agents are emerging, and where to implement them.
- **Graph RAG:** Graph RAG includes a graph database as a source of the contextual information in the RAG system. Existing RAG systems use lexical or semantic search to retrieve and send text chunks of documents to the LLM. Graph RAG offers additional capability, as it can also provide structured entity information to the LLM. Information stored as a graph can potentially provide richer context, combining the entity textual description with its many properties and relationships.
  - *How to prepare:* Another theme of this research is that effective information extraction and understanding are crucial. Graph techniques, with their ability to capture complex content and relationships, will become increasingly popular for organizations that wish to leverage RAG. See How to Evaluate the Applicability of Knowledge Graphs for Your Use Cases for examples of knowledge graph use cases.

- **Fine-tuning models:** Many organizations are looking to create customized LLMs that contain their own data and are optimized for specific use cases. Fine-tuning involves taking a pretrained LLM as a starting point and further training it on a new dataset.
  - *How to prepare:* Quick Answer: When to Fine-Tune Large Language Models provides guidance on where fine-tuning fits. Based on Gartner client experiences, fine-tuning approaches require a fair amount of effort to collect the necessary data. Not many clients that we've spoken with have successfully fine-tuned these models. Moreover, in some domains with specialized vocabulary, such as finance or healthcare, adopting fine-tuned embedding models before fine-tuned generation models may be necessary.

Gartner clients are early in their journey and must start with the basics of RAG system improvement. Don't prematurely jump into high-difficulty RAG optimizations before you build a working system that you can test and improve with feedback from real users.

Recommendations

- **Conduct more discovery to align with user needs and system requirements.** By conducting detailed user interviews, surveys and usability testing, AI architects and engineers can gather valuable insights that help craft a system tailored to real-world applications. Additionally, leveraging analytics to monitor how users interact with the system can provide ongoing feedback that helps continuously refine the system.

- **Improve three components of RAG: data ingestion, retrieval and generation.** AI architects must work on enhancing data ingestion and retrieval to ensure that the most relevant and accurate information is available for generating responses. To enable a RAG system that can handle scaling and increased data diversity, AI architects and engineers should consider techniques such as better metadata and relationship extraction, complex content understanding, intelligent chunking, small-to-big retrieval, and hybrid search.

- **Implement robust evaluation and iterative improvement processes.** To ensure continuous improvement of RAG systems, organizations must implement robust evaluation mechanisms and an iterative development process. Teams need to set up detailed evaluations and tests that enable them to understand whether their RAG system is relevant, precise, safe, performant and cost-effective. Tests and evaluations must also provide signals into how well the context recommendation system is working. Adopting a modular approach to system development and focusing on the most important upstream pieces can allow teams to improve individual components of the RAG system as they work toward making the overall system work well.

Conclusion

As an AI engineer or AI architect, you should take a pragmatic approach to RAG development, focused on feasible and simple upstream component optimization, discovery, evaluations, prompt templates and guardrails. Don't go down the path of agentic RAG, graph-based solutions or fine-tuning without a strong need and justification for those high-difficulty domains of AI engineering.

Evidence

What We Learned From a Year of Building With LLMs (Part I), O'Reilly.

Building Performant RAG Applications for Production, LlamaIndex.

AI Engineer, 15 November 2023, "Building Production-Ready RAG Applications: Jerry Liu" [Video], YouTube.

Patterns for Building LLM-based Systems & Products, eugeneyan.com.

AI Engineer, 2 November 2023, "Building Blocks for LLM Systems & Products: Eugene Yan" [Video], YouTube.

How to Choose the Right Embedding Model for Your RAG Application, MongoDB.

Chunking Strategies for LLM Applications, Pinecone.

W. Glantz, 12 RAG Pain Points and Proposed Solutions, Towards Data Science, via Medium.

Mastering RAG: How to Select an Embedding Model, Galileo.

Evaluating the Ideal Chunk Size for a RAG System Using LlamaIndex, LlamaIndex.

Chunking for RAG: Best Practices, Unstructured.

ChromaDB Cookbook | The Unofficial Guide to ChromaDB: Filters, Chroma.

What Is Graph RAG? Ontotext.

MTEB: Massive Text Embedding Benchmark, Hugging Face.

[1] Evaluating the Effectiveness of LLM-Evaluators (aka LLM-as-Judge), eugeneyan.com.

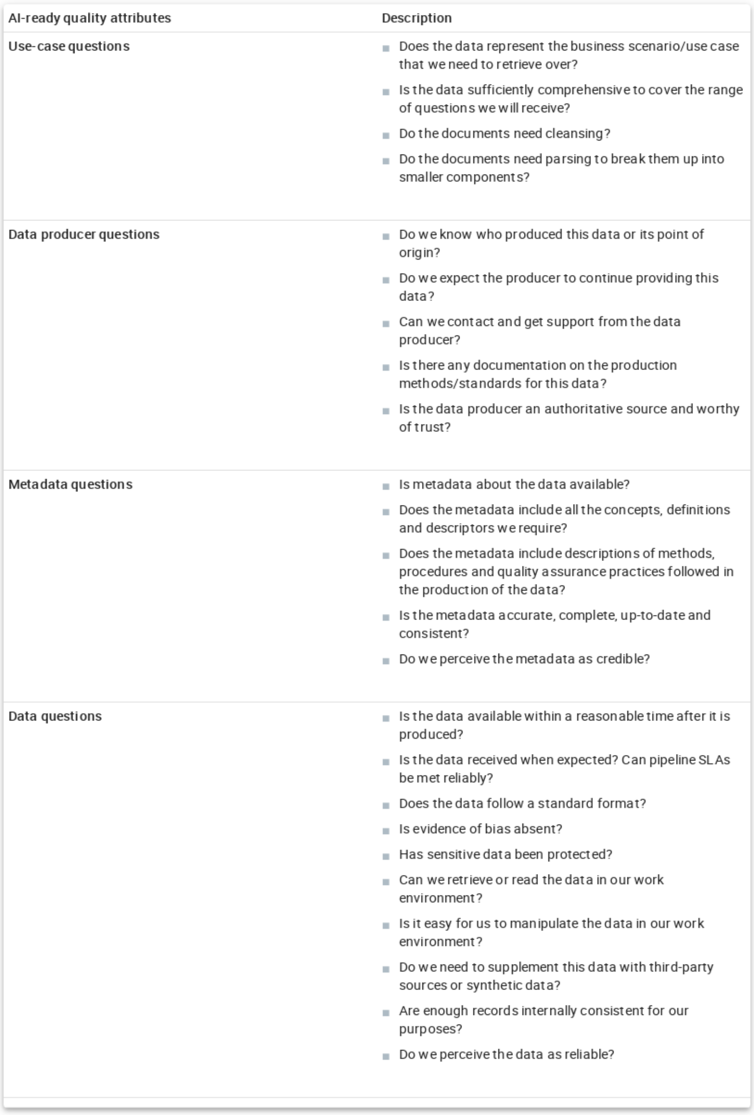
[2] Task-Specific LLM Evals That Do & Don't Work, eugeneyan.com.

Note 1: Quality Considerations for Unstructured Data

Table 2 provides a RAG-ready data quality checklist that describes a subset of quality dimensions for unstructured data.

Table 2: Quality Considerations for Unstructured Data

(Enlarged table in Appendix)



AI-ready quality attributes	Description
Use-case questions	■ Does the data represent the business scenario/use case that we need to retrieve over? ■ Is the data sufficiently comprehensive to cover the range of questions we will receive? ■ Do the documents need cleansing? ■ Do the documents need parsing to break them up into smaller components?
Data producer questions	■ Do we know who produced this data or its point of origin? ■ Do we expect the producer to continue providing this data? ■ Can we contact and get support from the data producer? ■ Is there any documentation on the production methods/standards for this data? ■ Is the data producer an authoritative source and worthy of trust?
Metadata questions	■ Is metadata about the data available? ■ Does the metadata include all the concepts, definitions and descriptors we require? ■ Does the metadata include descriptions of methods, procedures and quality assurance practices followed in the production of the data? ■ Is the metadata accurate, complete, up-to-date and consistent? ■ Do we perceive the metadata as credible?
Data questions	■ Is the data available within a reasonable time after it is produced? ■ Is the data received when expected? Can pipeline SLAs be met reliably? ■ Does the data follow a standard format? ■ Is evidence of bias absent? ■ Has sensitive data been protected? ■ Can we retrieve or read the data in our work environment? ■ Is it easy for us to manipulate the data in our work environment? ■ Do we need to supplement this data with third-party sources or synthetic data? ■ Are enough records internally consistent for our purposes? ■ Do we perceive the data as reliable?

Source: Gartner

Note 2: Large Context Window Models

We recommend using large context models to develop RAG prototypes. However, for production systems, we advise adhering to the tenet of passing only the necessary information. This approach optimizes for cost, performance and risk, leveraging RAG and its associated optimizations effectively. By balancing the use of large context windows with traditional RAG techniques, organizations can achieve scalable, efficient and secure AI-driven solutions.

Large context windows have some weaknesses organizations must consider:

1. **Cost-efficiency:** Utilizing large context windows increases operational costs, as organizations pay more for processing extensive amounts of context per prompt.

2. **Performance and latency:** Large input processing causes LLMs to perform slower than user expectations. By limiting the context to only what is necessary, organizations can enhance generation performance and reduce latency.

3. **Risk management:** Filling larger context windows with ever-greater amounts of context heightens the risk of including sensitive information or causing the model to overlook critical details buried in the context.

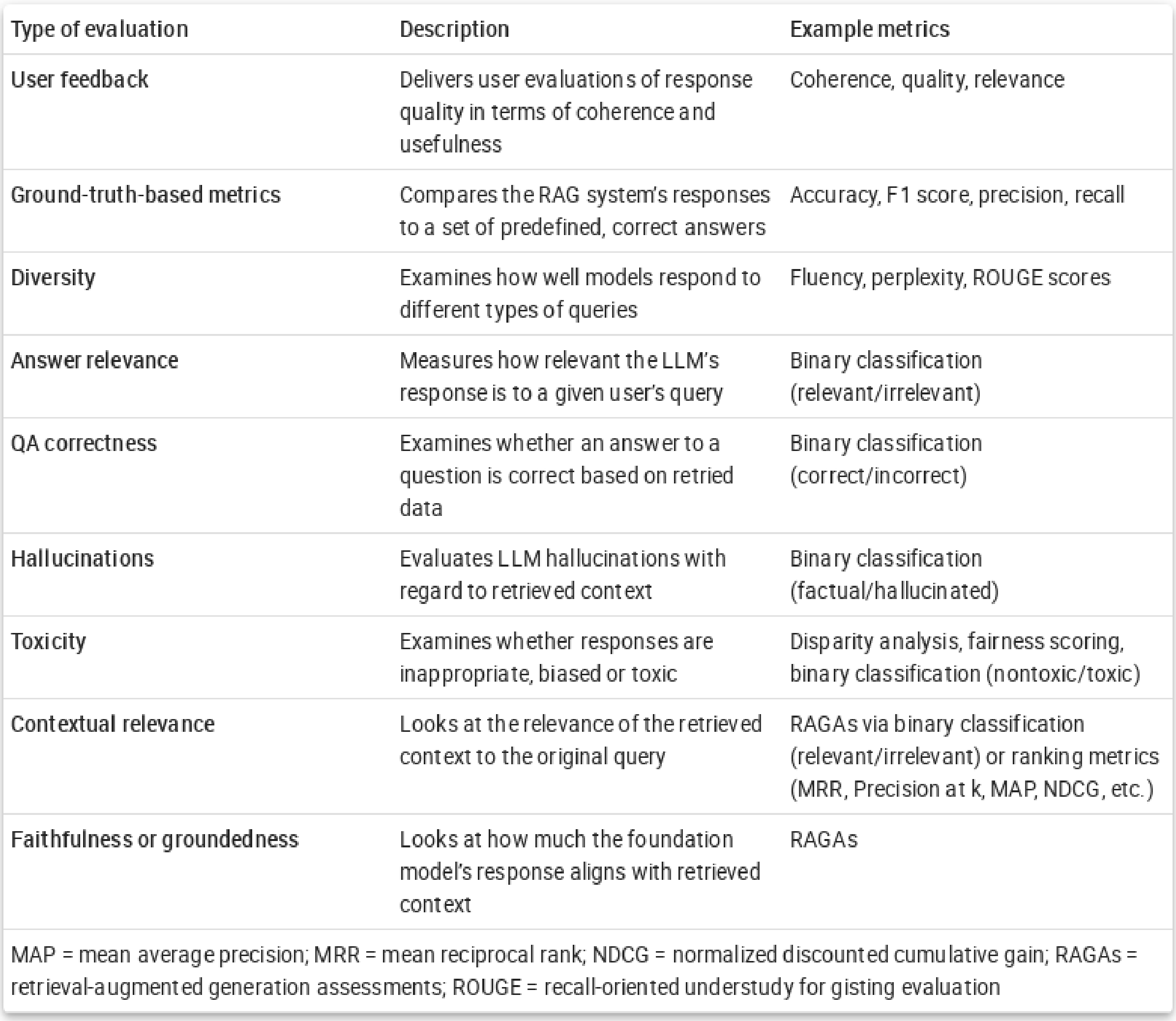
4. **Component limitations:** Not all components of a RAG system have a complete set of the features organizations may desire. For example, most embedding models don't support large context windows. This limits the applicability of large context features to specific workflows, such as bypassing the RAG system and directly inputting large context into the text prompt.

Note 3: RAG Quality Evaluations

Table 3 provides a sample of different metrics you can use to evaluate RAG system quality.

Table 3: RAG Quality Evaluations

(Enlarged table in Appendix)



Type of evaluation	Description	Example metrics
User feedback	Delivers user evaluations of response quality in terms of coherence and usefulness	Coherence, quality, relevance
Ground-truth-based metrics	Compares the RAG system's responses to a set of predefined, correct answers	Accuracy, F1 score, precision, recall
Diversity	Examines how well models respond to different types of queries	Fluency, perplexity, ROUGE scores
Answer relevance	Measures how relevant the LLM's response is to a given user's query	Binary classification (relevant/irrelevant)
QA correctness	Examines whether an answer to a question is correct based on retried data	Binary classification (correct/incorrect)
Hallucinations	Evaluates LLM hallucinations with regard to retrieved context	Binary classification (factual/hallucinated)
Toxicity	Examines whether responses are inappropriate, biased or toxic	Disparity analysis, fairness scoring, binary classification (nontoxic/toxic)
Contextual relevance	Looks at the relevance of the retrieved context to the original query	RAGAs via binary classification (relevant/irrelevant) or ranking metrics (MRR, Precision at k, MAP, NDCG, etc.)
Faithfulness or groundedness	Looks at how much the foundation model's response aligns with retrieved context	RAGAs

MAP = mean average precision; MRR = mean reciprocal rank; NDCG = normalized discounted cumulative gain; RAGAs = retrieval-augmented generation assessments; ROUGE = recall-oriented understudy for gisting evaluation

Source: Gartner

Recommended by the Author

Some documents may not be available as part of your current Gartner subscription.

Key Skills to Build LLMs With Retrieval-Augmented Generation

Getting Started With Retrieval-Augmented Generation

How to Engineer Effective Prompts for Large Language Models

2024 Planning Guide for Analytics and Artificial Intelligence

Best Practices for Building Successful AI Solutions

Use Model Guardrails to Regulate Generative AI Output and Behavior

10 Best Practices for Optimizing Generative AI Costs

How to Evaluate the Applicability of Knowledge Graphs for Your Use Cases

Introduce AI Observability to Supervise Generative AI

Quick Answer: Implementing Data Access Controls for Generative AI Assistants

---

Table 1: AI Feasibility Questions

Category	Questions
Data ingestion	
Data requirements	▪ What kind of data is necessary to build a knowledge corpus that addresses these questions? ▪ What is missing from our knowledge corpus?
Data complexity and processing	▪ How complex is the content, and what level of data preprocessing is required? ▪ Do we need more preprocessing to improve performance?
Retrieval	
Current capabilities	▪ What use cases are RAG systems currently handling effectively? ▪ Which parts of this system are not working effectively?
User queries	▪ What types of questions are typical from our users? ▪ Which questions are difficult for the system to answer?

Generation	
Evaluation methods	- How are we assessing the accuracy of the answers provided by the RAG system? - Are our users giving us valuable feedback?
Accuracy requirements	- What level of precision is necessary for the answers to be considered reliable? - What are the business/stakeholder expectations for this system?
Compliance and security	- What are the potential compliance, privacy and security risks associated with the data used by this system?

Source: Gartner

Table 2: Quality Considerations for Unstructured Data

AI-ready quality attributes	Description
Use-case questions	▪ Does the data represent the business scenario/use case that we need to retrieve over? ▪ Is the data sufficiently comprehensive to cover the range of questions we will receive? ▪ Do the documents need cleansing? ▪ Do the documents need parsing to break them up into smaller components?
Data producer questions	▪ Do we know who produced this data or its point of origin? ▪ Do we expect the producer to continue providing this data? ▪ Can we contact and get support from the data producer? ▪ Is there any documentation on the production methods/standards for this data? ▪ Is the data producer an authoritative source and worthy of trust?
Metadata questions	▪ Is metadata about the data available? ▪ Does the metadata include all the concepts, definitions and descriptors we require?

	■ Does the metadata include descriptions of methods, procedures and quality assurance practices followed in the production of the data? ■ Is the metadata accurate, complete, up-to-date and consistent? ■ Do we perceive the metadata as credible?
Data questions	■ Is the data available within a reasonable time after it is produced? ■ Is the data received when expected? Can pipeline SLAs be met reliably? ■ Does the data follow a standard format? ■ Is evidence of bias absent? ■ Has sensitive data been protected? ■ Can we retrieve or read the data in our work environment? ■ Is it easy for us to manipulate the data in our work environment? ■ Do we need to supplement this data with third-party sources or synthetic data? ■ Are enough records internally consistent for our purposes? ■ Do we perceive the data as reliable?

Source: Gartner

Gartner

Table 3: RAG Quality Evaluations

Type of evaluation	Description	Example metrics
User feedback	Delivers user evaluations of response quality in terms of coherence and usefulness	Coherence, quality, relevance
Ground-truth-based metrics	Compares the RAG system's responses to a set of predefined, correct answers	Accuracy, F1 score, precision, recall
Diversity	Examines how well models respond to different types of queries	Fluency, perplexity, ROUGE scores
Answer relevance	Measures how relevant the LLM's response is to a given user's query	Binary classification (relevant/irrelevant)
QA correctness	Examines whether an answer to a question is correct based on retried data	Binary classification (correct/incorrect)
Hallucinations	Evaluates LLM hallucinations with regard to retrieved context	Binary classification (factual/hallucinated)
Toxicity	Examines whether responses are inappropriate, biased or toxic	Disparity analysis, fairness scoring, binary classification (nontoxic/toxic)
Contextual relevance	Looks at the relevance of the retrieved context to the original query	RAGAs via binary classification (relevant/irrelevant) or ranking metrics (MRR, Precision at k, MAP, NDCG, etc.)
Faithfulness or groundedness	Looks at how much the foundation model's response aligns with retrieved context	RAGAs

MAP = mean average precision; MRR = mean reciprocal rank; NDCG = normalized discounted cumulative gain; RAGAs = retrieval-augmented generation assessments; ROUGE = recall-oriented understudy for gisting evaluation

Source: Gartner