Writing best practices to optimize RAG applications

AWS Prescriptive Guidance

AWS Prescriptive Guidance: Writing best practices to optimize RAG applications

Table of Contents

Writing best practices to optimize RAG applications

Ivan Cui and Samantha Stuart, Amazon Web Services

Large language models (LLMs) have revolutionized the field of artificial intelligence with their remarkable ability to understand and generate human-like text. However, they face a significant limitation: they can only work with knowledge contained in their training data. This is where Retrieval Augmented Generation (RAG) helps. It offers a solution that combines LLMs with external knowledge sources, such as your organization's data and documents. Through a two-stage process that involves information retrieval and response generation, RAG enables AI systems to access and incorporate up-to-date information from various sources, resulting in more accurate and informed responses that bridge the gap between static model knowledge and dynamic real-world information needs.

How can you optimize content for retrieval in a RAG-based application? This guide provides best practices to help you optimize the formatting and writing style of text-based content in the knowledge base. Optimizing the content enhances the context that helps RAG applications understand task-specific information more accurately. When the system retrieves highly relevant and accurate content, then the quality of the LLM's response improves. Optimizing the context delivery process at a system level is called *context engineering*, and it forms an essential part of agentic RAG architectures. In agentic RAG, one or more additional LLMs reason and act on intake requests before the RAG execution. This facilitates a multi-step information delivery process. As RAG architectures grow increasingly complex, source content optimization remains the most direct means of delivering clear context to LLMs. These best practices are designed to help you maximize your organization's investment in a RAG application.

Intended audience

This guide is intended for AI engineers, data scientists, data engineers, or software developers who are building LLM applications with one or more RAG components. To understand the concepts and recommendations in this guide, you should be familiar with vector databases and prompts for LLMs.

Objectives

The recommendations in this guide can help you achieve the following:

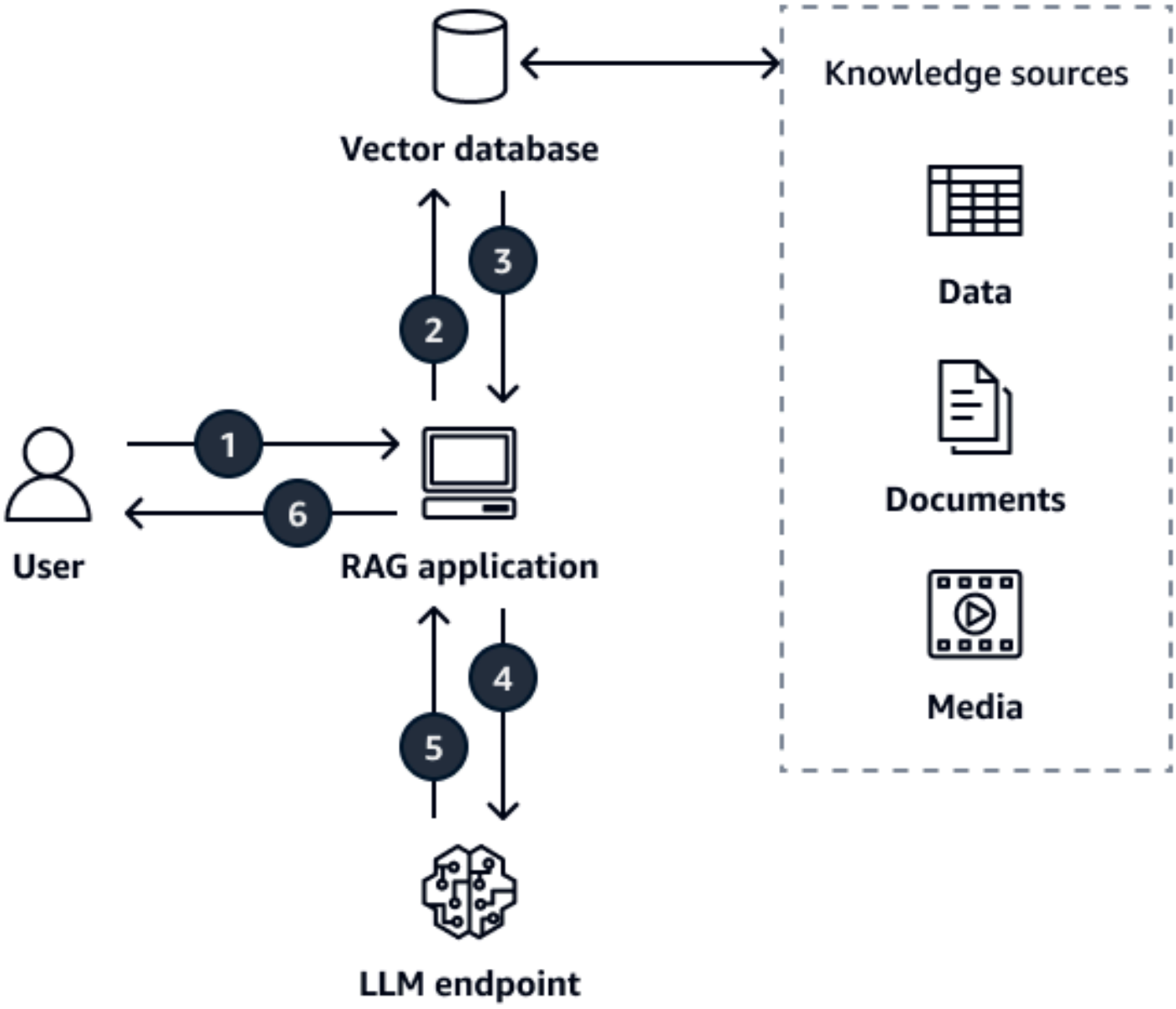
- Improve the accuracy and relevancy of responses generated by RAG applications by providing well-structured and semantically rich source documents, optimized for token usage and redundancy.
- Help RAG applications to better understand domain-specific knowledge and context by providing clear definitions and explanations within source documents.
- Facilitate easier maintenance and knowledge base updates for RAG applications by adhering to consistent formatting and structuring guidelines across source documents.
- Improve the scalability of RAG solutions by breaking down large, monolithic documents into smaller, self-contained units that can be efficiently indexed and retrieved.

Understanding LLMs and RAG

To understand how enhancing source document quality enhances the quality of a RAG response, you must understand the internal workings of an LLM. The true power of LLMs lies in their ability to use self-attention mechanisms and transformer architectures. These advanced techniques enable the models to effectively process and relate different parts of the input sequence, regardless of their position or distance within the text. This capability is a stark contrast to traditional language models, which often struggle with long-range dependencies and context understanding. Furthermore, LLMs are trained on an unprecedented scale. Some of the largest models are comprised of trillions of parameters and have ingested terabytes of textual data from diverse sources. This massive scale allows LLMs to develop a rich understanding of language, capturing subtle nuances, idioms, and contextual cues that were previously challenging for AI systems. The result is a class of models that can generate coherent and fluent text and demonstrate remarkable capabilities in tasks such as question answering, text summarization, and even code generation.

To use these models, we can turn to services such as Amazon Bedrock, which provides access to a variety of foundation models from Amazon and third-party providers, including Anthropic, Cohere, and Meta. You can use Amazon Bedrock to experiment with state-of-the-art models, customize and fine-tune them, or incorporate them into your generative AI-powered solutions through a single API.

Although LLMs excel at capturing patterns and generating coherent text, they often lack access to up-to-date or specialized information. RAG combines the generative power of LLMs with a retrieval component that can access and incorporate relevant information from external sources, as part of the materialized LLM prompt. Examples of external sources include Knowledge Bases for Amazon Bedrock, intelligent search systems such as Amazon Kendra, or vector databases such as Amazon OpenSearch Service.



The diagram describes the following workflow:

1. The user submits a query to the RAG application.
2. The RAG application queries a vector database that contains knowledge sources, such as documents, data, or media.
3. The RAG application retrieves the relevant information from the vector database based on semantic similarities between the query and stored documents.
4. The RAG application augments the original prompt with the retrieved context and sends it to the LLM endpoint.

5. The LLM endpoint generates a response and returns it to the RAG application.
6. The RAG application returns the generated response to the user.

At its core, RAG employs a two-stage process. In the first stage, a retrieval model identifies and retrieves relevant documents or passages based on the input query. This retrieval model can be a traditional information retrieval system, a dense retrieval model, or a combination of both. In the second stage, the retrieved information and the original query are fed into an LLM as a fully materialized prompt template. LLMs depend heavily on the quality of the source content delivered by the retriever component. They apply a self-attention mechanism to mathematically encode how the retrieved content relates to the task. The LLM then generates a response based on both the query and the retrieved information. In RAG, controlling the quality of the retrieved source documents represents a direct means of improving an LLM's internal representation of a task. RAG effectively augments the LLM's training data with relevant external data. This approach allows RAG to take advantage of the strengths of both LLMs and retrieval systems, enabling the generation of more accurate and informed responses that incorporate current and specialized knowledge.

Vectors and embeddings

Vectors and embeddings are fundamental concepts in machine learning and natural language processing. *Vectors* are mathematical objects that represent quantities that have both magnitude and direction. In the context of natural language processing (NLP), words, sentences, or documents are often represented as vectors in high-dimensional vector spaces. *Embeddings*, on the other hand, are a way to represent objects like words or documents in a lower-dimensional vector space where the relationships between the vectors capture semantic or syntactic similarities. Word embeddings, for instance, allow words with similar meanings to have similar vector representations. This helps algorithms to understand and process language more effectively.

Vector databases

In generative AI, a *vector database* is a database that stores and manages vector representations of documents, queries, or other objects. It is designed to efficiently store and retrieve vectors. This supports fast and scalable operations such as semantic search and similarity matching. Vector databases index vectors by using specialized data structures, such as Hierarchical Navigable Small World (HNSW) graphs or K-Nearest Neighbors (KNN) algorithms. These data structures allow for fast nearest-neighbor searches, making it possible to quickly find similar vectors in the database.

Semantic search

Semantic search is a technique that improves the relevance of search results by understanding the intent and context of the query, rather than just matching keywords. In technical terms, semantic search involves comparing the vector representations of the query and the documents in the database to find the most relevant matches. Different retrieval strategies can be used for semantic search, including but not limited to:

- **HNSW** – A graph-based data structure that organizes vectors in a way that makes it efficient to search for nearest neighbors.
- **KNN** – An algorithm that finds the K closest vectors to a query vector based on a distance metric, such as cosine similarity.
- **Cosine similarity** – A measure of similarity between two non-zero vectors that measures the cosine of the angle between them. It is often used in semantic search to compare the direction of vectors in a high-dimensional space.
- **Locality-sensitive hashing (LSH)** – A technique that hashes similar vectors to the same or nearby buckets with high probability. This allows for approximate nearest-neighbor searches, which can be faster than exact searches in high-dimensional spaces.

Challenges in source data that affect RAG applications

One of the significant challenges in developing an optimal Retrieval-Augmented Generation (RAG) application lies in the nature of the raw data or documents used. Often, enterprises use existing documents that were created for human reference. These documents often include hyperlinks and image screenshots to promote understanding. However, these elements obstruct semantic retrieval due to excerpt token limits. This results in poor retriever performance.

The following are the most common raw document challenges for an optimal RAG application:

- **Lack of structured formatting and metadata** – Raw documents can lack clear section headings, subheadings, or metadata. This makes it challenging to identify and extract relevant information. For example, a long document without clear headings can make it difficult to determine the context of specific information.
- **Informal and inconsistent language** – Raw documents often contain informal language or inconsistent terminology. This can confuse RAG models. For instance, abbreviations that are not defined in the document or already known by the LLM might be used throughout a document.
- **Verbosity and redundancy** – Raw documents may be verbose and contain unnecessary or redundant information. This can overwhelm RAG models, leading to less concise and relevant responses. Examples include a document that repeats the same information multiple times or multiple documents that contain similar or contradictory information.
- **Ambiguous terms and phrases** – Raw documents can contain ambiguous terms or phrases that might be interpreted in multiple ways. This ambiguity can lead to misinterpretation by RAG models and inaccurate responses. For example, a document that uses a term with multiple meanings can result in a response that does not align with the intended meaning.
- **Injection of graphic and hyperlink elements** – Raw documents that contain graphics and hyperlink information work well for human consumption. However, these elements can consume the retrieval token limit. The result is that excerpts might be incomplete. For example, graphics and hyperlink URLs are returned as part of the retrieval, which uses up the retrieval tokens, and key information from subsequent paragraphs is missing.
- **Lack of domain-specific knowledge or context** – Raw documents can lack the necessary domain-specific knowledge or context required for accurate generation. This can limit the ability of RAG models to generate relevant and accurate responses. An example is a document that references specialized concepts without providing context. This might lead to responses that are not meaningful in the given domain.

Although this list isn't comprehensive, it provides a starting point for enterprises to think about what is not working and why. Documents might have one or more of these challenges. The key to optimizing a RAG application is to use a set of documents that adhere to writing best practices that optimize retrieval.

Documentation best practices for RAG applications

Developing a successful Retrieval-Augmented Generation (RAG) application requires careful consideration of various document-related factors to optimize its performance. The best practices in this section are curated based on experience building RAG systems with many organization leaders. The following are several key best practices for documents to enhance the effectiveness of your RAG application:

- **Use headings and subheadings properly** – Organizing your content with clear headings and subheadings improves readability and helps RAG models understand the structure of your documents. This practice enables the models to better navigate and extract information from the documents, which enhances the quality of the generated responses.
- **Ensure numbering is sequential** – When using numbered lists, it's important to maintain proper numbering to avoid confusion. Ensure that each list item is numbered sequentially without skipping numbers. This helps maintain clarity and coherence in your content.
- **Add transitions between list items** – Providing transitions between items in a bulleted or numbered list helps guide the LLM through the content. For example, you can use phrases like "After completing step 2, do..." to connect ideas and improve the flow of information.
- **Replace tables** – Avoid using tables. Format this information in multi-level bulleted lists or in a flat-level syntax. *Flat-level syntax* is arranging elements or items at the same hierarchical level, without nested levels of subordination. These structures help LLMs to digest the information. Because most indexed documents are read from left to right, flat-level syntax allows information to follow more coherently without needing to reference an additional dimension. This format is more conducive to RAG applications because it presents information in a structured and easily digestible manner.
- **Preprocess graphical information for efficiency** – Multi-modal LLMs can ingest both image and text. Reduce the resolution of images, remove redundant images, and describe the content of graphical elements in text format. These measures improve meaningful context, avoid consuming tokens unnecessarily, and improve accessibility for RAG models.
- **Add session starters for common queries** – When addressing common questions or tasks, such as "How do I order software?", add a session starter that transitions the reader into the process. For example, you might add "If you are looking to order software, follow the steps below…". This helps to create high semantic matching, which helps the LLM construct a cohesive response.
- **Add summarization to each section** – After each heading or subheading, add a brief and concise summary of the content in that section. This can increase semantic coverage and reinforce

key points. This improves the accuracy of similarity search within the embedding space, thus improving the performance of the RAG application. This is particularly helpful if the document is intended for both LLM and human consumption or if table and graphical elements are necessary.

- **Disambiguation** – Documents should be concise and focused. LLMs generate responses based on retrieved excerpts, so disambiguation helps the model use clear and relevant information. This results in more accurate and informative responses.
- **Define abbreviations and set context** – LLMs are trained on large amount of internet data, and most of the time, they do not have the context of an enterprise's internal documents. Therefore, setting context, defining abbreviations, and avoiding or defining company-specific terminology helps the LLM understand your enterprise data. This helps the LLM to answer questions more accurately and can help prevent hallucinations.
- **Restructure large documents into smaller documents for efficient tagging and indexing** – Avoid indexing a large document that contains multiple subtopics. Consider dividing the large document into smaller, self-contained documents that have clear titles. This improves indexing and tagging.

FAQ

Why is it important to optimize documents for RAG applications?

Raw documents are often written for human consumption without considering the requirements of advanced AI systems, such as Retrieval Augmented Generation (RAG) applications. Optimizing documents by following best practices can significantly improve the performance and accuracy of RAG applications by providing structured, unambiguous, and relevant information to the models.

What are some common challenges with raw documents that can hinder RAG performance?

Some key challenges include lack of structured formatting and metadata, informal or inconsistent language, verbosity and redundancy, ambiguous terms and phrases, inclusion of hyperlink elements, and lack of domain-specific context. These issues can confuse RAG models and lead to inaccurate or irrelevant responses. For more information, see Challenges in source data that affect RAG applications in this guide.

How can the use of headings and subheadings improve RAG performance?

Clear headings and subheadings help RAG models understand the structure and context of the content. This enables them to better navigate and extract relevant information from the documents and improves the quality of generated responses. For more information, see Documentation best practices for RAG applications in this guide.

Why is it recommended to replace table information with flat-level syntax?

It can be challenging for RAG models to interpret tables because they require an understanding of the two-dimensional structure. Presenting table information in a flat-level syntax or bulleted list helps models to more easily process the information, leading to better performance. For more information, see Documentation best practices for RAG applications in this guide.

How can adding summaries enhance RAG performance?

Including concise summaries at the beginning of each section or subsection can increase semantic coverage and reinforce key points. This improves the accuracy of similarity searches within the embedding space, which ultimately enhances the performance of the RAG application. For more information, see Documentation best practices for RAG applications in this guide.

Why is it important to define abbreviations and set context for LLMs?

LLMs are trained on a broad range of data, but they lack context for enterprise-specific abbreviations or terminology. Defining abbreviations and providing context helps LLMs understand and respond more accurately. This can help prevent hallucinations or misinterpretations. For more information, see Documentation best practices for RAG applications in this guide.

Next steps and resources

This guide discusses a set of document-level challenges for RAG applications and best practices to mitigate them. These learnings are curated from interviewing and discussing with industry leaders, and they are backed by enterprise use cases.

To begin optimizing your documents for RAG applications, we recommend conducting an audit of your existing documents. Identify areas that pose challenges to the RAG application. Examples include a lack of structure, ambiguous language, or excessive use of graphical elements. Prioritize documents that are frequently accessed or critical to your business operations. Collaborate with subject matter experts to implement the best practices in this guide. Make sure that documents are restructured with clear headings, concise language, and context-setting elements. For new documents, establish guidelines and templates that ensure consistency and help authors adhere to the best practices. Additionally, consider investing in tools or services that can automate aspects of the document optimization process, such as using generative AI to restructure documents. By taking a proactive approach to document optimization, you can unlock the full potential of RAG applications and drive more accurate and insightful results across your organization.

The following resources can help you understand and build RAG applications in your organization.

Resources

AWS documentation

- Choosing an AWS vector database for RAG use cases (AWS Prescriptive Guidance)
- Deploy a RAG use case on AWS by using Terraform and Amazon Bedrock (AWS Prescriptive Guidance)
- Develop advanced generative AI chat-based assistants by using RAG and ReAct prompting (AWS Prescriptive Guidance)
- Retrieve data and generate AI responses with Amazon Bedrock Knowledge Bases (Amazon Bedrock documentation)
- Retrieval Augmented Generation (Amazon SageMaker AI documentation)
- Retrieval Augmented Generation options and architectures on AWS (AWS Prescriptive Guidance)

Other AWS resources

- Create a multimodal assistant with advanced RAG and Amazon Bedrock (AWS blog post)

Document history

The following table describes significant changes to this guide.

Change	Description	Date
Initial publication	—	July 24, 2025