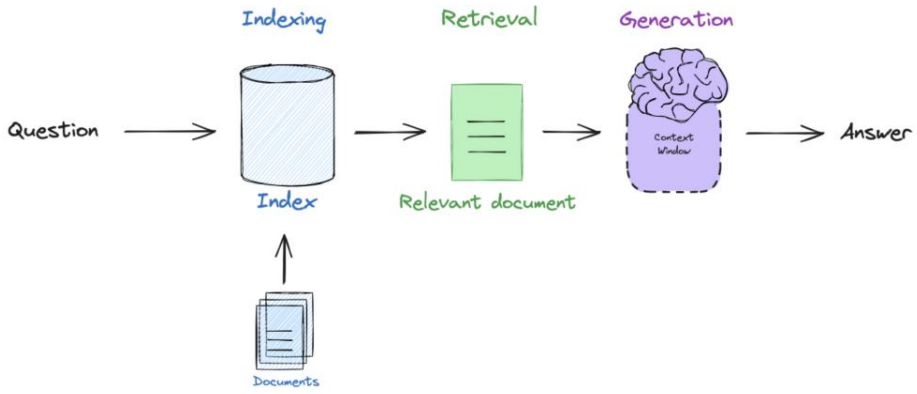
Table of Contents

Introduction

Retrieval Augmented Generation (RAG) is a technique that enhances language models by combining them with external knowledge retrieval.

RAG integrates a retrieval system with a generative language model. When given a query, it first retrieves relevant information from a knowledge base, then uses this information to generate a more informed and accurate response.



Source: LangChain

❖ Key components of RAG

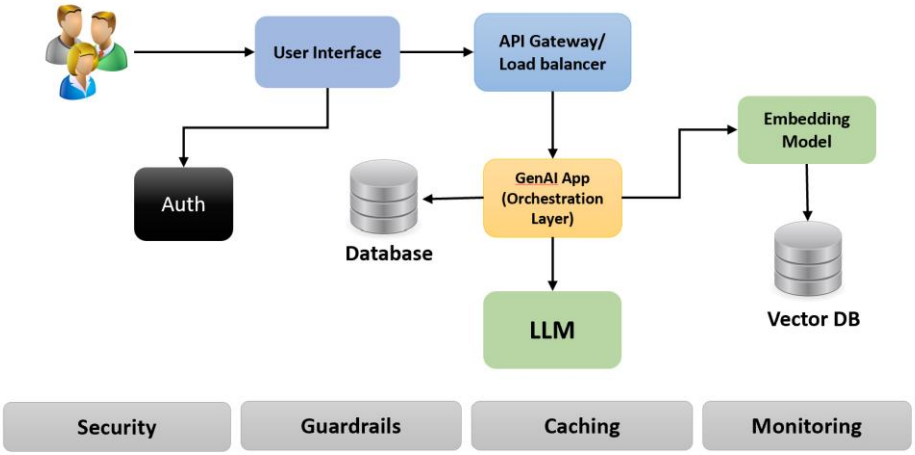
- ✓ **Knowledge base:** The external data source serves as the repository of information that the RAG system can rely on.
- ✓ **Retrieval system:** The retrieval system typically consists of two main parts:
  - **A vector database** to efficiently store and search through embeddings
  - **An embedding model** to convert queries and documents into vector representations
- ✓ **Language model:** RAG systems use Large Language Models (LLMs) to generate responses based on the retrieved information and the original query.

RAG is essential for building accurate and context-aware GenAI applications. As queries become more complex and information increases, basic RAG may not be enough. Advanced RAG techniques, including agentic RAG, improve the **retrieval** and **generation** process, leading to more relevant, accurate, and adaptable responses.

Let’s first understand what is Agentic and Non-Agentic based GenAI Applications.

❖ Non-Agentic GenAI Applications

Non-Agentic GenAI applications refers to the creation and improvement of generative AI systems that *do not possess agency or autonomous decision-making capabilities*. These AI models are designed to generate content or assist with tasks based on their training data and user inputs, without having their own goals, motivations, or ability to act independently.



Let's break it down:

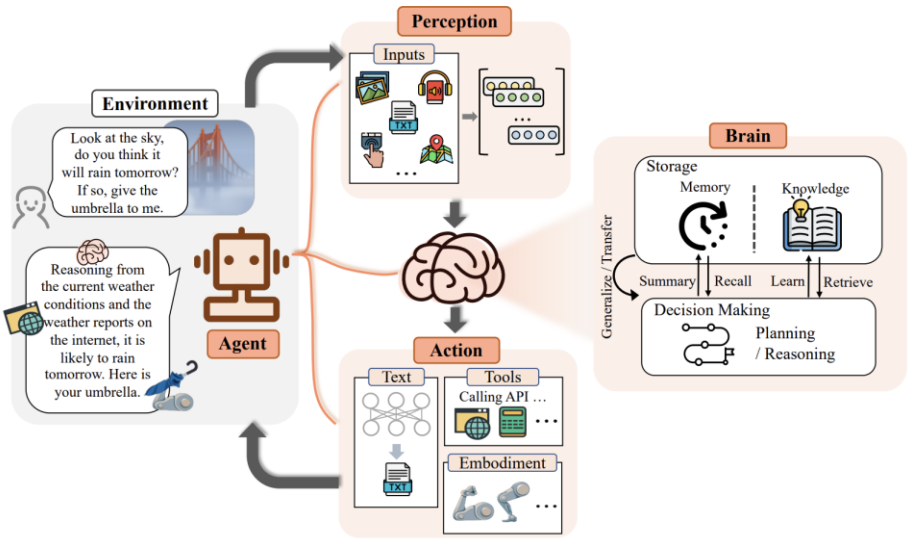
- **Generative AI (GenAI):** These are AI systems that can create new content, such as text, images, or audio, based on patterns learned from training data.
- **Non-Agentic:** This means the AI doesn't have agency, it doesn't make decisions on its own or pursue its own goals. It simply responds to user inputs and generates outputs based on its training.
- **Example:** Imagine a GenAI system designed to help with writing stories. This non-agentic GenAI would work as follows:
  - ✓ **User input:** The user provides a prompt like "Write a short story about a dog who learns to fly."
  - ✓ **AI processing:** The AI processes this request using its trained understanding of language, storytelling, and the concepts of dogs and flying.

- ✓ **Output generation:** The AI generates a story based on the prompt, without making any autonomous decisions about the plot or characters beyond what's implied by the user's request.
- ✓ **User interaction:** The user can refine the story by giving more specific instructions, which the AI will follow without developing its own preferences or goals for the story.

In this example, the GenAI is non-agentic because it doesn't decide on its own to write stories, choose topics, or make creative decisions beyond what it's explicitly instructed to do.

❖ Agentic-based GenAI Applications

Agentic-based LLM Development refers to creating language models that can act more autonomously, make decisions, and pursue goals, rather than just responding to prompts.



Source: 2309.07864 (arxiv.org)

Let's break it down:

- ➢ **Tools -** Agentic LLMs can be equipped with the ability to use external tools or APIs. For example:

- ✓ Web search capabilities to find up-to-date information
- ✓ Access to calculators for complex math
- ✓ Ability to call coding environments to test and run code
- ✓ Integration with databases or knowledge bases

Example: An agentic LLM tasked with market research could autonomously decide to use a web search tool to gather recent data, then use a spreadsheet tool to analyse the information.

- **Planning -** Agentic LLMs can break down complex tasks into steps and create plans to achieve goals. This involves:
  - ✓ Understanding the overall objective
  - ✓ Identifying necessary sub-tasks
  - ✓ Prioritizing actions
  - ✓ Adjusting the plan as new information becomes available

  Example: Given the task "Write a comprehensive report on renewable energy trends," an agentic LLM might plan to: **1)** Research recent developments **2)** Analyse data on adoption rates **3)** Identify key players in the industry **4)** Draft report sections **5)** Synthesize conclusions.

- **Memory -** Agentic LLMs can be designed with various types of memory to enhance their capabilities:
  - ✓ Short-term memory: Retaining context within a conversation
  - ✓ Long-term memory: Storing and retrieving information from past interactions or learned experiences
  - ✓ Episodic memory: Remembering specific events or examples

  Example: An agentic LLM with advanced memory could recall previous conversations with a user, remember their preferences, and use this information to provide more personalized responses over time.

- **Reasoning -** Agentic LLMs can employ various reasoning strategies to make decisions and solve problems:
  - ✓ Deductive reasoning: Drawing conclusions from given information.
  - ✓ Inductive reasoning: Making generalizations from specific observations.
  - ✓ Abductive reasoning: Forming the most likely explanation from incomplete information.

  Example: When analysing a complex issue, an agentic LLM could use reasoning to evaluate different perspectives, identify logical fallacies, and arrive at well-supported conclusions.

- **Goal-Oriented Behaviour -** Agentic LLMs can be given overarching goals or objectives that guide their actions:
  - ✓ Pursuing multiple sub-goals to achieve a larger objective
  - ✓ Balancing competing priorities
  - ✓ Recognizing and resolving conflicts between goals

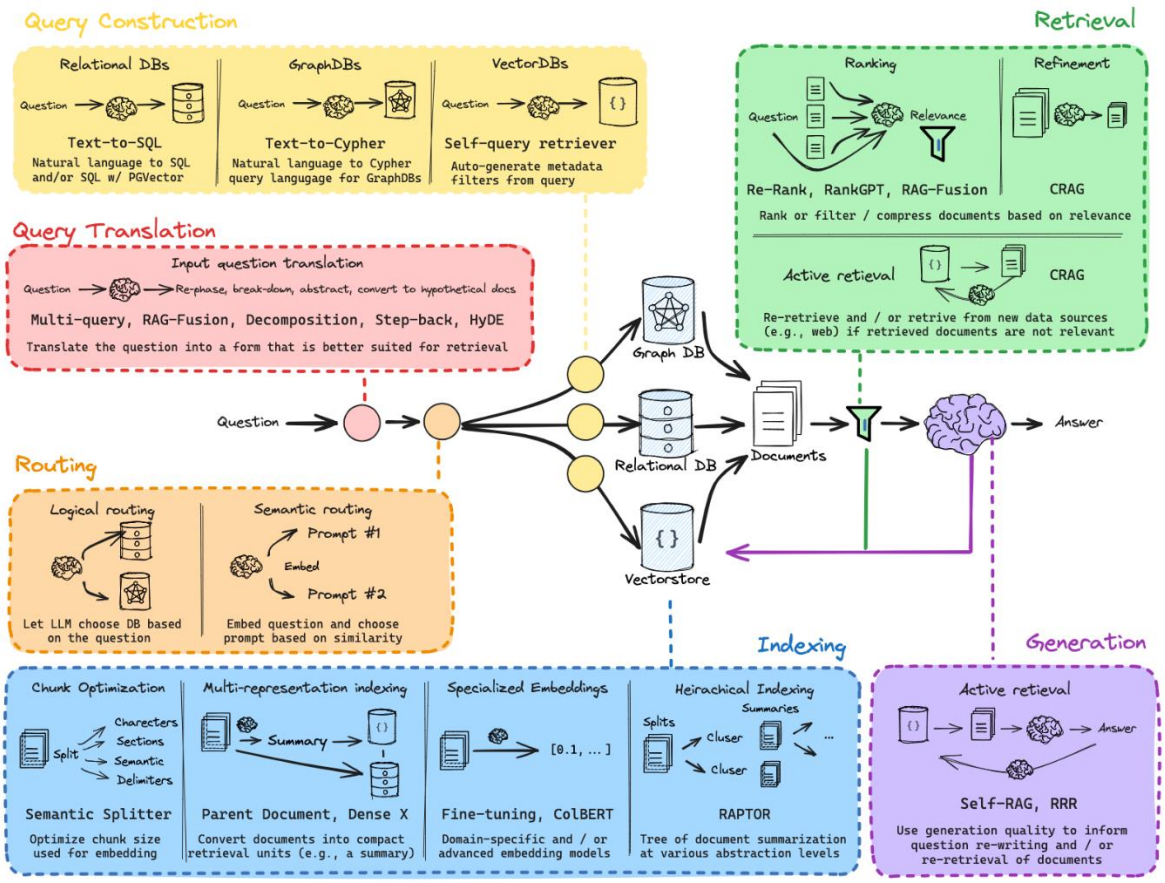
  Example: An agentic LLM designed to assist with academic research might have goals like "*ensure factual accuracy,*" "*identify gaps in current knowledge*," and "*suggest novel research directions.*"

- **Self-Improvement -** Agentic LLMs could potentially have mechanisms for self-improvement:
  - ✓ Identifying areas where their knowledge or performance is lacking
  - ✓ Seeking out new information to fill knowledge gaps
  - ✓ Refining their decision-making processes based on outcomes

  Example: After completing a task, an agentic LLM might analyse its performance, identify areas for improvement, and adjust its approach for similar future tasks.

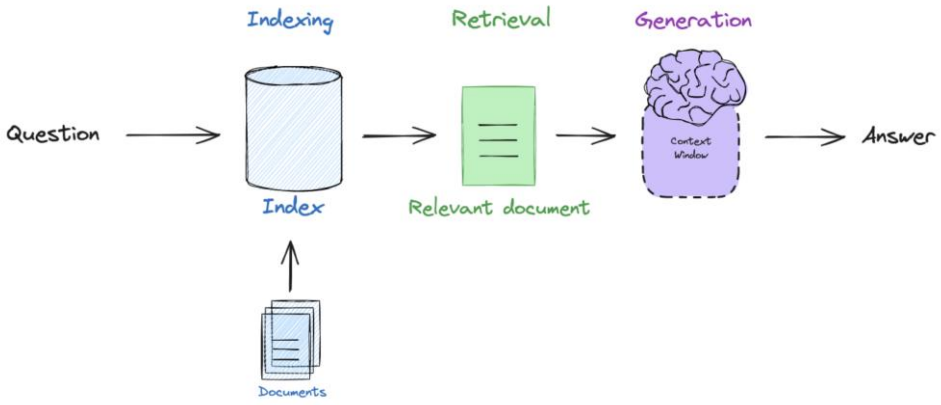
We will explore different advanced RAG techniques that can improve the performance of our GenAI systems. Let's explore each of these techniques in detail, showing how to implement them using LangChain and providing code examples.

RAG Flow by LangChain



Source: LangChain AI Doc

Basic RAG



Source: [LangChain]

- **[GitHub Notebook Link]**
- ### What is Basic RAG?

  Basic RAG is the standard, straightforward implementation of Retrieval-Augmented Generation. It involves retrieving relevant information from a knowledge base in response to a query, then using this information to generate an answer using a language model.

- ### Why we need RAG?

  - Combines the broad knowledge of language models with specific, up-to-date information
  - Improves accuracy of responses by grounding them in retrieved facts
  - Reduces hallucinations common in standalone language models
  - Allows for easy updates to the knowledge base without retraining the entire model

- ### How it works?

  - As mentioned in **Introduction.**

- ### Use case

  A streamlined knowledge base and retrieval system that is well-organized and easy to navigate. It features well-formatted source data with minimal ambiguity, allowing for the efficient generation and retrieval of high-quality elements. By utilizing an

embedding model and vector database, we can enhance the retrieved data, which serves as input for a large language model to generate accurate responses.

Re-ranking

Source: LlamaIndex

- **GitHub Notebook Link**
- ### What is Re-ranking?

  Re-ranking in RAG is a critical process that refines and reorders the initially retrieved information before it's fed into a generative AI model. It acts as a smart filter, ensuring that the most relevant and high-quality content is prioritized for the generation task.

  Key aspects:

  - ✓ **Relevance optimization:** Improves the quality of information used by the LLM.
  - ✓ **Intelligent sorting:** Uses advanced algorithms to reassess and reorder retrieved passages.
  - ✓ **Context consideration:** Takes into account the query intent and user context.
  - ✓ **Integration point:** Sits between retrieval and generation components in the RAG pipeline.

  By effectively re-ranking retrieved information, RAG systems can significantly enhance the accuracy, relevance, and overall quality of the generated AI responses.

- ### Why we need Re-ranking?

  - ➢ **Relevance optimization:** Improves the quality of information used by the generative model, leading to more accurate and contextually appropriate outputs.

- **Reducing noise:** Helps filter out less relevant or potentially misleading information that might negatively impact the generated content.
- **Efficiency:** By prioritizing the most relevant information, it allows the generative model to focus on the most important content, potentially improving response time and reducing computational overhead.
- **Handling complex queries:** Enables the RAG system to better address nuanced or multi-faceted queries by ensuring a diverse yet relevant set of retrieved information.

❖ How it works?

- **Initial Retrieval:** Fetch the ***top-k*** documents based on vector similarity.
- **Re-ranking:** Apply advanced models or criteria to reorder these k documents.
- **Selection:** Choose the ***top-n*** re-ranked documents for generation.

❖ Use Case

Re-ranking is useful in advanced RAG systems where initial search results need improvement. For example, after retrieving a list of documents, re-ranking can analyse these results based on relevance to the original query, user preferences, or contextual information. This process enhances the order of the results, ensuring that the most relevant documents are shown at the top. By refining the output, re-ranking leads to more accurate and user-friendly information retrieval.

❖ Re-ranking Techniques

Cross-Encoder Models

- ✓ Use transformer models to compare query-document pairs directly.
- ✓ More accurate but takes more computational power.

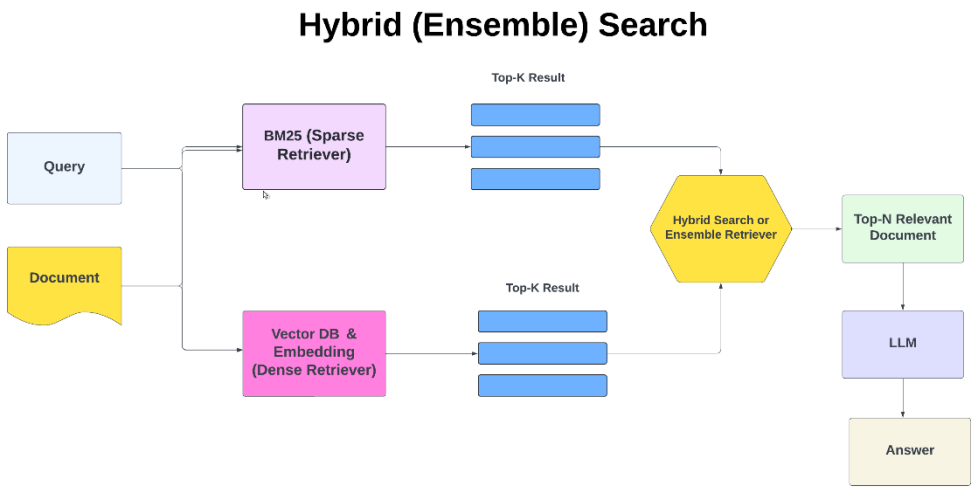
Learning to Rank (LTR)

- ✓ Use machine learning models trained on relevance judgments.
- ✓ Can consider various features like click-through rates (CTRs) and document freshness.

Query-specific Re-ranking

- ✓ Change the re-ranking strategy based on the query type or intent.

Hybrid (Ensemble) Search



❖ **GitHub Notebook Link**

❖ What is Hybrid Search?

Hybrid (Ensemble) Search is a technique that combines multiple search methods to improve retrieval performance. Typically, it combines traditional keyword-based search (like BM25) with semantic search (using embedding models). This approach can provide better results than either method alone, especially for queries that have both ***keyword-specific*** and ***semantic aspects.***

❖ Why we use Hybrid Search?

- **Improved accuracy:** Hybrid search combines multiple search techniques, leveraging the strengths of each to achieve better overall results.
- **Versatility:** It can handle diverse types of queries and content more effectively than a single search method.
- **Robustness:** By not relying on a single approach, hybrid search is less vulnerable to the weaknesses of any particular method.

- **Adaptability:** Can be tailored to specific use cases or domains by adjusting the combination of search techniques.
- **Better handling of complex queries:** Especially useful for queries that require both semantic understanding and specific keyword matching.

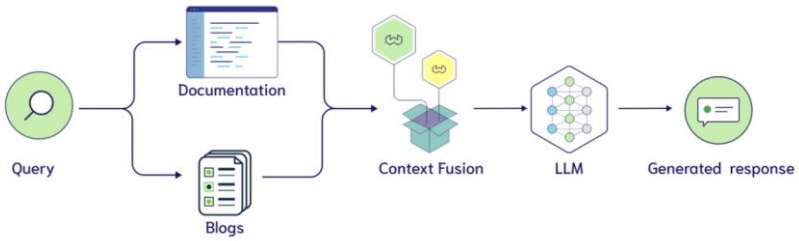
❖ How it works?

- **Combine search methods:** Integrate different search techniques such as keyword-based, semantic, and vector search.
- **Weighting system:** Assign importance to results from each method based on the query type or specific use case.
- **Parallel processing:** Run multiple search methods simultaneously to improve speed.
- **Result aggregation:** Merge and deduplicate results from different search methods.
- **Ranking algorithm:** Develop a unified ranking system that considers scores from all employed search methods.

❖ Use Case

The Hybrid (Ensemble) Search works well when different retrievers can help each other. For example, it combines a sparse retriever like BM25, which is good at keyword searches, with a dense retriever that finds similar documents using embeddings. This mix takes advantage of the keyword matching from the sparse retriever and the deeper understanding from the dense retriever, creating a stronger retrieval system.

Multi-index



Source: [Erica Cardenas]

- **GitHub Notebook Link**

- ## What is Multi-index?

  Multi-index is a RAG technique that involves using multiple separate indexes or databases to store and retrieve information. Instead of having a single, monolithic index, data is distributed across several specialized indexes. Each index can be optimized for different types of data or query patterns, allowing for more efficient and targeted

- ## Why we need Multi-index?

  - **Multi- Faster retrieval:** Multi-index allows Generative AI to quickly access relevant information from large knowledge bases, reducing response time.
  - **Better context handling:** Enables the AI to efficiently navigate different topics or domains, improving contextual understanding and relevance.
  - **Scalability:** Supports growing knowledge bases without sacrificing performance, essential for evolving Generative AI systems.
  - **Improved accuracy:** By accessing more focused and relevant information, the AI can generate more accurate and contextually appropriate responses.
  - **Flexibility:** Allows for easy updates and additions to specific areas of knowledge without affecting the entire system, keeping the AI current.

- ## How it works?

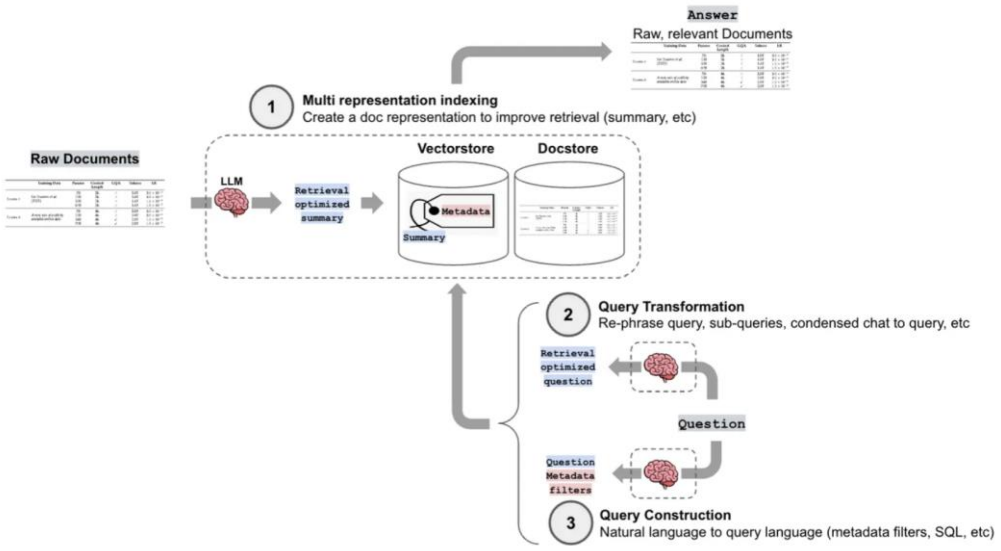
  - **Split knowledge:** Divide the AI's knowledge into smaller, topic-based parts.
  - **Create multiple indexes:** Make a separate index for each part of the knowledge.
  - **Smart query routing:** Send each user question to the most relevant index(es).

- **Search multiple indexes:** Look through several relevant indexes at once when needed.
- **Combine results:** Merge and rank information found in different indexes.
- **Update regularly:** Keep adding new information to the right indexes.
- **Use in AI generation:** Feed the retrieved information into the AI to create responses.
- **Improve over time:** Learn from what works best and adjust the system accordingly.

❖ Use Case

A big e-commerce site uses a multi-index method for product searches. They keep separate indexes for product descriptions, customer reviews, and technical specs. When a user searches, the system checks all these indexes at the same time. The description index finds general matches, the review index shows user opinions, and the specs index focuses on detailed features. This approach gives users more complete and relevant search results, enhancing overall search quality and satisfaction.

Query Expansion/Transformation



Source: [LangChain Blog]

❖ [GitHub Notebook Link]

❖ What is Query Expansion/Transformation RAG?

Query Expansion is a technique used to improve the effectiveness of information retrieval systems by reformulating or augmenting the original query. The goal is to improve the recall of relevant documents by including related terms or concepts that might not have been explicitly mentioned in the original query.

❖ Why we need Query Expansion?

- **Enhanced understanding:** Helps Generative AI better interpret user intent by considering related terms and concepts.
- **Improved retrieval:** Increases the chances of finding relevant information, even when the user's query is vague or incomplete.
- **Handling ambiguity:** Allows the AI to explore multiple interpretations of a query, leading to more comprehensive responses.
- **Vocabulary gap bridging:** Addresses differences between user language and the AI's knowledge base terminology.
- **Contextual relevance:** Enables the AI to consider broader context, resulting in more nuanced and accurate responses.

❖ How it works?

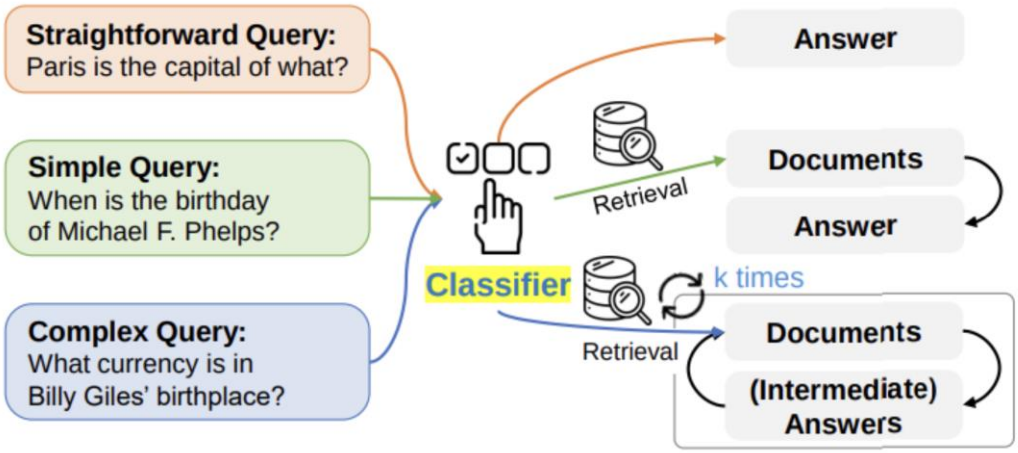
- **Synonym expansion:** Add synonyms of key terms in the user's query to broaden the search.
- **Semantic analysis:** Use Natural Language Processing to identify related concepts and expand the query accordingly.
- **Word embedding:** Utilize word vector representations to find semantically similar terms.
- **Knowledge graph integration:** Leverage structured relationships between concepts to expand queries with related terms.
- **User context consideration:** Incorporate user history or preferences to personalize query expansion.
- **Feedback loop:** Analyse successful responses to refine and improve future query expansions.

- **Language model utilization:** Employ pre-trained language models to suggest relevant query expansions.
- **Domain-specific expansion:** Apply industry or topic-specific terminologies for more accurate expansion in specialized fields.

❖ Use Case

A medical research database uses Query Expansion to help researchers find relevant studies. For example, if a researcher searches for "heart attack treatment," the system adds related medical terms like "myocardial infarction therapy" and "coronary thrombosis management," as well as concepts like "stents" and "rehabilitation." This broader search retrieves more relevant studies, including those that use different terms or focus on specific aspects of treatment. As a result, researchers can access a wider range of useful information they might have missed with their original search.

Adaptive-RAG



Source: https://blog.lancedb.com/adaptive-rag/

❖ **GitHub Notebook Link**

❖ What is Adaptive RAG?

Adaptive-RAG is an advanced technique that improves upon traditional RAG systems by dynamically adapting the retrieval process based on the specific query and context. This

approach aims to enhance the relevance and accuracy of retrieved information, leading to better-generated responses.

❖ Why we need Adaptive RAG?

- **Dynamic optimization:** Allows Generative AI to adjust its retrieval strategy based on query complexity and context.
- **Improved accuracy:** Enhances the relevance of retrieved information, leading to more precise and contextually appropriate responses.
- **Handling diverse queries:** Better equipped to address a wide range of query types, from simple to complex.
- **Continuous improvement:** Enables the system to learn and refine its retrieval strategies over time, enhancing overall performance.

❖ How it works?

- **Query analysis:** Assess the complexity and nature of each incoming query.
- **Strategy selection:** Choose the most appropriate retrieval method based on the query analysis.
- **Dynamic indexing:** Adjust the indexing approach in real-time to suit the current query requirements.
- **Feedback incorporation:** Use the success of previous responses to inform future retrieval strategies.
- **Context awareness:** Consider user context and session history to refine the retrieval process.
- **Performance monitoring:** Continuously track and analyse the effectiveness of different retrieval strategies.
- **Machine learning integration:** Employ ML models to predict the most effective retrieval approach for each query type.

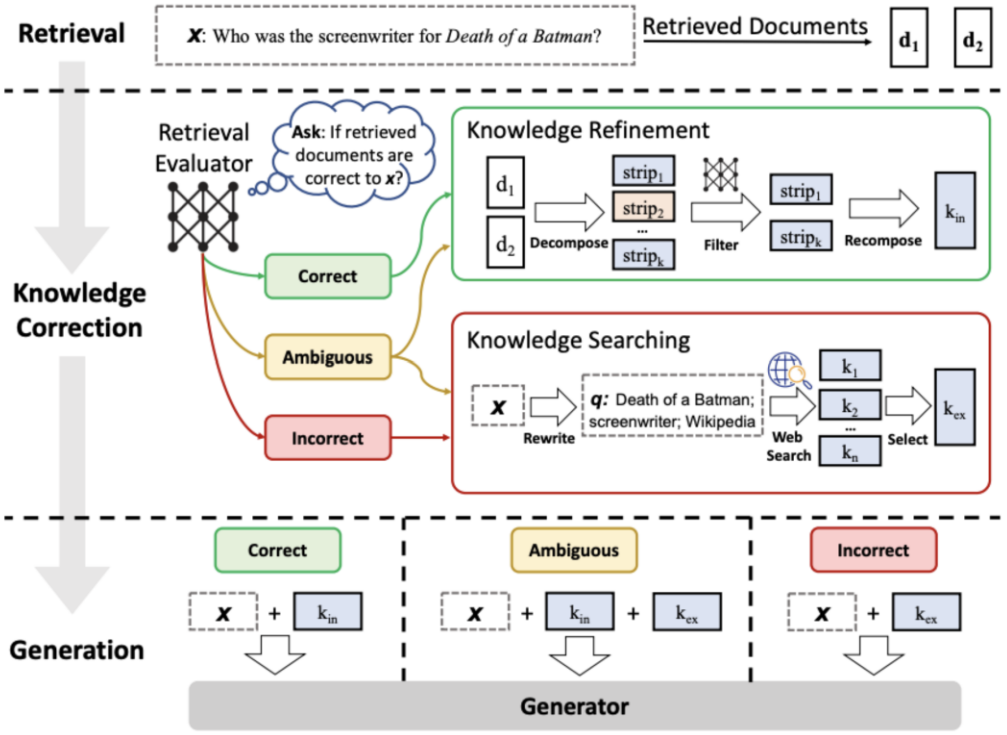
❖ Use Case

A large customer support chatbot uses Adaptive-RAG to manage different customer questions. When a customer asks a simple question about store hours, the system quickly finds the answer with a basic method. For more complex issues, like technical

support for a product malfunction, the system switches to a more advanced approach, using multiple data sources and a better language model. This way, the chatbot can provide fast answers for simple questions and detailed help for complex ones, keeping customers satisfied while using resources efficiently

---

Corrective-RAG



Source: https://blog.langchain.dev/agentic-rag-with-langgraph/

- **GitHub Notebook Link**
- ### What is Corrective-RAG?

  Corrective RAG is a technique that introduces an additional step to verify and correct the information retrieved before generating the final response. This method aims to reduce errors and inconsistencies in the generated output by cross-checking the retrieved information against known facts or trusted sources. It often involves a separate model or module dedicated to fact-checking and error correction.

❖ Why we need Corrective RAG?

- **Accuracy improvement:** Helps Generative AI identify and correct potential errors or inconsistencies in retrieved information.
- **Reliability enhancement:** Increases the trustworthiness of AI-generated responses by validating information before use.
- **Handling conflicting data:** Enables the AI to reconcile contradictory information from different sources.
- **Bias mitigation:** Assists in identifying and correcting potential biases in the retrieved information.
- **Continuous learning:** Allows the system to improve its knowledge base over time by identifying and correcting inaccuracies.

❖ How it works?

- **Query analysis:** Assess the complexity and nature of each incoming query.
- **Strategy selection:** Choose the most appropriate retrieval method based on the query analysis.
- **Dynamic indexing:** Adjust the indexing approach in real-time to suit the current query requirements.
- **Feedback incorporation:** Use the success of previous responses to inform future retrieval strategies.
- **Multi-stage retrieval:** Implement a cascading approach, starting with simpler methods and progressing to more complex ones as needed.
- **Context awareness:** Consider user context and session history to refine the retrieval process.

❖ Use Case

A medical information system uses Corrective RAG to give patients reliable health information. When a user asks about the side effects of a medication, the system first retrieves relevant details from its database. Before answering, the Corrective RAG checks this information against the latest medical guidelines. If it finds any outdated or incorrect details, it updates them. For example, if a new side effect has been discovered,

the system adds that information. This ensures patients get the most accurate and current medical information, which is essential for their health and safety.

Self-Adaptive RAG

- **GitHub Notebook Link**

- ### What is Self-Adaptive RAG?

  Self-Adaptive RAG is an advanced technique that autonomously optimizes its performance over time. It uses machine learning algorithms to continuously analyse its own outputs, user feedback, and performance metrics to refine its retrieval and generation strategies. This system can adjust its parameters, update its knowledge base, and modify its decision-making processes without constant human intervention, allowing it to adapt to changing information landscapes and user needs.

- ### Why we need Self-Adaptive RAG?

  - **Continuous optimization:** Enables the RAG system to automatically improve its performance over time without manual intervention.
  - **Adaptability to changing data:** Allows the system to adjust to evolving knowledge bases and query patterns.
  - **Personalization:** Can tailor its behaviour to individual users or specific use cases, enhancing relevance.
  - **Efficiency gains:** Optimizes resource usage by adapting retrieval strategies based on their effectiveness.
  - **Robustness: I**mproves the system's ability to handle a diverse range of queries and content types.

- ### How it works?

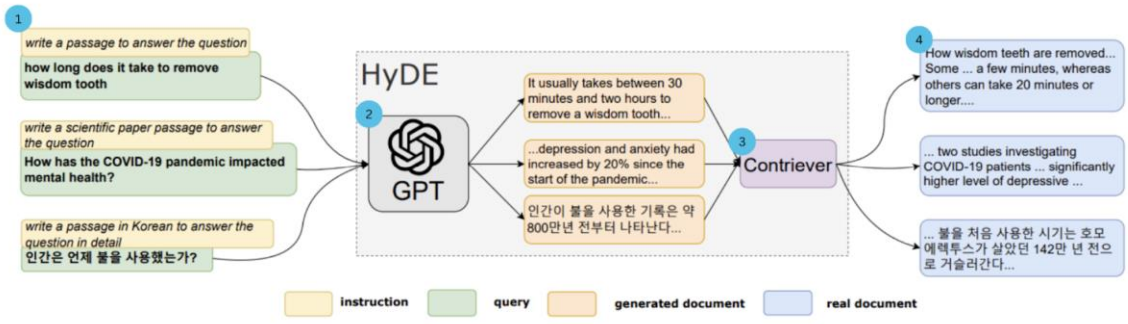
  - **Fact-checking:** Implement automated fact-checking mechanisms to verify retrieved information against reliable sources.
  - **Cross-referencing:** Compare information from multiple retrieved sources to identify consistencies and discrepancies.
  - **Confidence scoring:** Assign confidence scores to retrieved information based on source reliability and corroboration.

- **Error detection:** Use Natural Language Processing to identify potential logical inconsistencies or factual errors.
- **Feedback incorporation:** Learn from user feedback and corrections to improve future retrieval and correction processes.
- **Adaptive learning:** Adjust retrieval and correction strategies based on patterns of identified errors and corrections.

❖ Use Case

A large news recommendation system uses Self-Adaptive RAG to give users personalized news articles. It starts with a basic model for retrieving news recommendations. As users engage with the articles, the system analyses factors like click rates and reading time. Over time, it learns user preferences, like favouring local news in the morning and global news in the evening. It can also adjust the summaries it generates, providing shorter ones for mobile users and longer ones for desktop users. Additionally, the system updates its knowledge base with new topics and adjusts the importance of news sources based on their reliability. This self-adaptive approach keeps the recommendations relevant and effective, reducing the need for manual updates.

Hypothetical Document Embeddings (HDE)



Source: [Zilliz Cloud]

❖ [GitHub Notebook Link]

❖ What is HDE?

Hypothetical Document Embeddings (HDE) is an advance RAG technique that generates synthetic document embeddings based on the query, rather than relying solely on existing document embeddings. This approach creates a "hypothetical" perfect document that would ideally answer the query, and then uses this embedding to retrieve the most similar actual documents from the database. HDE aims to bridge the gap between query intent and document content, especially for complex or nuanced queries.

❖ Why we need HDE?

- **Query enhancement:** Enables Generative AI to better understand and represent complex or hypothetical queries.
- **Improved retrieval:** Allows for more accurate matching between queries and existing documents, even for very new scenarios.
- **Creativity support:** Facilitates the AI's ability to explore and generate responses for future-oriented questions.
- **Context expansion:** Helps bridge the gap between user intent and available information in the knowledge base.
- **Handling edge cases:** Improves the GenAI performance on queries that fall outside the scope of existing documents.

❖ How it works?

- **Query-to-document transformation:** Convert complex queries into hypothetical document-like structures.
- **Embedding generation:** Create vector representations of these hypothetical documents using the same embedding model as the knowledge base.
- **Similarity matching:** Use these embeddings to find the most relevant existing documents in the vector space.
- **Synthetic document creation:** Generate temporary, context-rich documents based on the query for improved matching.
- **Dynamic weighting:** Adjust the importance of hypothetical embeddings based on query complexity and available information.

- **Iterative refinement:** Use initial results to further refine the hypothetical document and improve retrieval.
- **Integration with language models:** Leverage Large Language Models to expand queries into more comprehensive hypothetical documents.
- **Performance tracking:** Monitor and analyse the effectiveness of hypothetical embeddings to continually improve the process.

❖ Use Case

HDE improves Imagine a movie recommendation system. You ask for "a funny family movie with talking animals and adventure."

- HDE creates an imaginary perfect movie matching your description.
- It then finds real movies most similar to this imaginary one.
- You might get suggestions like:
  - "Madagascar"
  - "Zootopia"
  - "The Secret Life of Pets"

Even if these movies don't use all your exact words in their descriptions, HDE understands the concept you're looking for and finds the best matches available.

Best Practices

❖ Pre-processing (data cleaning, chunking etc.)

Pre-processing in the context of RAG (Retrieval Augmented Generation) is a crucial step that prepares your data for efficient retrieval and effective use by the language model.

Let's break it down:

- **Data Cleaning**

  Data cleaning involves preparing your raw data to ensure it's in a consistent, usable format. This step is crucial because real-world data often contains

inconsistencies, errors, or irrelevant information that can negatively impact the performance of your RAG system.

Examples of data cleaning tasks:

- ✓ **Removing irrelevant information**
  - Stripping out HTML tags from web-scraped content
  - Removing headers, footers, or boilerplate text from documents
- ✓ **Handling special characters and encoding issues**
  - Converting all text to UTF-8 encoding
  - Replacing or removing non-printable characters
- ✓ **Standardizing text**
  - Converting all text to lowercase (or maintaining case where appropriate)
  - Standardizing date formats (e.g., converting all dates to YYYY-MM-DD)
- ✓ **Correcting obvious errors**
  - Fixing common misspellings
  - Standardizing company names or product names
- ✓ **Removing or replacing sensitive information**
  - Anonymizing personal data
  - Replacing specific numbers or identifiers with placeholders
  - **Example:** Raw text: " ACME Corp. (founded on 01/05/1995) is a leading provider of widgets. " Cleaned text: "acme corp founded on 1995-01-05 is a leading provider of widgets"

➢ Chunking

- ✓ Chunking involves breaking down large documents or texts into smaller, more manageable pieces. This is important because:
  - Most embedding models have a token limit
  - Smaller chunks allow for more precise retrieval
  - It helps in providing context without overwhelming the model with irrelevant information

- ✓ **Chunking strategies**

  - **Useful Article**
    - LangChain Text Splitters Strategies
    - 5 Level of Text Splitting
  - **Fixed-size chunking**
    - Splitting text into chunks of a fixed number of tokens or characters
    - Example: Splitting a document into chunks of 512 tokens each
  - **Sentence-based chunking**
    - Keeping complete sentences together
    - Example: Ensuring each chunk ends at a sentence boundary
  - **Paragraph-based chunking**
    - Keeping related sentences together
    - Example: Using paragraphs as natural chunk boundaries
  - **Semantic chunking**
    - Using NLP techniques to identify and keep together semantically related content
    - Example: Using topic modelling to group related paragraphs
  - **Overlap chunking**
    - Creating chunks with some overlap to maintain context
    - Example: Each chunk includes the last 50 tokens of the previous chunk

- ➢ **Chunking example**
  - ✓ **Original text:** "Artificial Intelligence (AI) is transforming various industries. Machine Learning, a subset of AI, focuses on creating systems that learn from data. Deep Learning, a part of Machine Learning, uses neural networks with many layers."
  - ✓ **Chunked (with overlap):**
    - **Chunk 1:** "Artificial Intelligence (AI) is transforming various industries. Machine Learning, a subset of AI, focuses on creating systems that learn from data."

- **Chunk 2:** "Machine Learning, a subset of AI, focuses on creating systems that learn from data. Deep Learning, a part of Machine Learning, uses neural networks with many layers."

✓ **In practice, you would typically combine these steps:**

- Clean the Raw Data
- Perform initial chunking
- Further clean or process the chunks if necessary
- Generate embeddings for each chunk
- Store the chunks and their embeddings in your vector database

This pre-processing ensures that when your RAG system needs to retrieve information, it's working with clean, well-formatted, and appropriately sized pieces of text, leading to more accurate and relevant retrievals.

❖ Post-processing (result filtering, fact-checking)

Post-processing in **RAG** refers to the steps taken after the language model has generated its initial response. The goal is to improve the quality, accuracy, and relevance of the final output.

Let's break it down into two main components:

➢ Result Filtering

Result filtering involves refining the generated content to ensure it meets specific criteria or quality standards. This step helps remove irrelevant or low-quality information from the final output.

Key aspects of result filtering include:

✓ **Relevance scoring**

- Compute a relevance score between the generated content and the original query
- Filter out sections with low relevance scores

- ✓ **Confidence thresholding**
  - If the model provides confidence scores, filter out parts of the response below a certain threshold
- ✓ **Length-based filtering**
  - Remove excessively short or long responses that are likely to be incomplete or off-topic
- ✓ **Keyword-based filtering**
  - Ensure the response contains key terms related to the query
  - Filter out sections with irrelevant keywords
- ✓ **Sentiment or tone filtering**
  - For applications requiring a specific tone, filter out content that doesn't match the desired sentiment
  - **Example:**

    Query: "What are the health benefits of green tea?"

    Generated response: "Green tea has numerous health benefits. It's rich in antioxidants, may boost brain function, and can help with weight loss. Green tea is made from Camellia sinensis leaves. The production process involves steaming the leaves to prevent oxidation."

    Filtered response: "Green tea has numerous health benefits. It's rich in antioxidants, may boost brain function, and can help with weight loss."

    (The information about production is filtered out as it's less relevant to the health benefits query.)

➢ Fact-checking

Fact-checking involves verifying the accuracy of the generated content against trusted sources or the original retrieved documents. This step is crucial for maintaining the reliability and trustworthiness of the system.

Key aspects of fact-checking include:

- ✓ **Source verification**
  - Cross-reference generated facts with the original retrieved documents

- Flag or remove information not supported by the source material

✓ **Numerical consistency**

- Check if numerical values in the response match those in the source documents
- Flag significant discrepancies for human review

✓ **Named entity verification**

- Verify that names of people, places, or organizations are correctly spelled and referenced

✓ **Temporal consistency**

- Ensure that dates and time-related information are accurate and consistent with the source material

✓ **External API fact-checking**

- For critical applications, use external fact-checking APIs or databases to verify key claims

✓ **Contradiction detection**

- Identify and flag any internal contradictions within the generated response
- **Example:**

  Retrieved document: "The Eiffel Tower was completed in 1889 and stands at a height of 324 meters."

  Generated response: "The Eiffel Tower, completed in 1888, is 330 meters tall."

  Fact-checked response: "The Eiffel Tower, completed in 1889 [CORRECTED], is 324 meters tall [CORRECTED]."

  Practical implementation often involves:

  - Generate the initial response using RAG
  - Apply result filtering to remove irrelevant or low-quality content
  - Perform fact-checking on the filtered response
  - Flag any inconsistencies or potential inaccuracies
  - Either automatically correct verified inaccuracies or present the flagged content for human review
  - Produce the final, filtered, and fact-checked response

By implementing these post-processing steps, you can significantly improve the quality, accuracy, and reliability of your RAG system's outputs. This is especially important in domains where factual accuracy is crucial, such as healthcare, finance, or legal applications.

❖ Continuous evaluation and tuning the process

This practice is more comprehensive and ongoing. It includes:

- **Evaluating the entire RAG pipeline** This means assessing not just the model's performance, but also the retrieval quality, the relevance of retrieved information, and the overall output quality.
- **Fine-tuning multiple components** This could involve adjusting:
  - ✓ The retrieval mechanism (e.g., tweaking embedding models or retrieval algorithms)
  - ✓ The reranking system (if used)
  - ✓ The prompts used for the language model
  - ✓ The post-processing steps
- **Iterative improvements** Based on ongoing evaluations, you might update your document chunking strategy, adjust your indexing method, or refine your query processing.
- **Model fine-tuning** Yes, this can include periodic fine-tuning of the language model itself, especially if you're using a smaller model that you can fine-tune in-house.

❖ Version Control for Data and Model (LLMOPs sort of things)

Version control in the context of RAG systems involves tracking changes and managing different versions of both the data used for retrieval and the models used for generation. This practice is essential for ensuring reproducibility, tracking performance improvements, and managing updates to your AI system.

Few key things to check:

- **Data Version Control**
  - ✓ This involves tracking changes to your knowledge base or document collection used for retrieval.

- ✓ **Tech Stack:** DVC
- ✓ **Importance**
  - Ensures reproducibility of results
  - Allows rolling back to previous versions if new data introduces issues
  - Helps in analysing the impact of data changes on system performance
- ✓ **Implementation strategies**
  - Git-based version control for textual data
  - Specialized data version control tools like DVC (Data Version Control)
  - Metadata tagging of documents with version information
- ✓ **Example using DVC**

```
from dvc import api

# Retrieve a specific version of your data
with api.open('data/knowledge_base.txt', rev='v1.0') as f:
    kb_v1 = f.read()

# Compare with current version
with api.open('data/knowledge_base.txt') as f:
    kb_current = f.read()

# Analyze differences
# ...
```

➢ Model Version Control

- ✓ This involves tracking different versions of your language models, embedding models, and any other models used in your RAG pipeline.
- ✓ **Tech Stack:** MLflow, AWS Sagemaker MLOPs etc..
- ✓ **Importance**
  - Allows easy rollback to previous model versions if issues arise
  - Facilitates A/B testing of different model versions
  - Ensures reproducibility of results across different model iterations
- ✓ **Implementation strategies**
  - Using model registries (like MLflow, Weights & Biases)
  - Containerization of model versions (like using Docker)
  - Systematic naming conventions and metadata tagging
- ✓ **Example using MLflow:**

```python
import mlflow

# Log model version
with mlflow.start_run():
    mlflow.log_param("model_name", "llama3.1")
    mlflow.log_param("model_version", "v2")
    # ... train or fine-tune model ...
    mlflow.sklearn.log_model(model, "rag_model")

# Later, load a specific model version
model_name = "rag_model"
model_version = "1"
model = mlflow.pyfunc.load_model(
    model_uri=f"models:/{model_name}/{model_version}"
)
```

➢ Combined Data and Model Versioning

- ✓ In RAG systems, it's important to track the compatibility between data versions and model versions.
- ✓ **Implementation strategies**
  - Creating "experiment" or "release" versions that combine specific data and model versions
  - Using configuration management tools to define and version entire system setups
- ✓ **Example using a configuration file**

```yaml
# config.yaml
version: '2.1'
data:
  knowledge_base: 'v3.2'
  embedding_index: 'v1.5'
models:
  language_model: 'llama3.1-v2'
  embedding_model: 'sentence-transformers/all-MiniLM-L6-v2'
retriever:
  type: 'hybrid'
  version: 'v1.1'
```

➢ Continuous Integration/Continuous Deployment (CI/CD) for RAG

- ✓ Implement automated testing and deployment pipelines that are version-aware.
- ✓ **Example workflow**
  - New data or model version is pushed

- Automated tests run on the new version
- If tests pass, the new version is tagged and potentially deployed
- Performance metrics are logged for the new version
- Monitoring and Logging: Implement comprehensive logging that includes version information for all components.

✓ **Example log entry**

```
2023-07-28 14:30:15 | Query: "What is climate change?" |
Data Version: v3.2 | Model Version: llama3.1-v2 |
Retriever Version: v1.1 | Response: "Climate change refers to..."
```

➢ **Version control best practices**

✓ **Regular backups:** Ensure all versioned data and models are regularly backed up.

✓ **Documentation:** Maintain clear documentation of what changes between versions.

✓ **Validation:** Implement validation checks when switching between versions to ensure compatibility.

✓ **Access control:** Implement proper access controls to manage who can make changes to different versions.

By implementing robust version control for both data and models in your RAG system, you can:

1. Easily track and reproduce results.
2. Quickly identify and roll back problematic changes.
3. Facilitate collaboration among team members.
4. Ensure compliance with any regulatory requirements for traceability.
5. Streamline the process of testing and deploying improvements to your system.