Agentic Multi-Retrieval Augmented Generation Chatbot

Agentic Multi-RAG Chatbot Project Report

Prepared by:

1. Manan Arora
2. Nitin Malik
3. Dhananjay Sharma
4. Siva Venkata Pavan Kumar Vempati
5. Lokesh Kaushik
6. Abraham Alejandro Saenz Tirado

August 21, 2025

Submitting to: Robin Manchanda

Robin.Manchanda@kinectrics.com

Table of Contents

1. Introduction

This report provides a comprehensive analysis and detailed explanation of an agentic retrieval-augmented generation (RAG) pipeline, as demonstrated in the accompanying codebase. The pipeline represents a state-of-the-art system designed to facilitate advanced document-grounded question answering. It integrates multiple key components into a cohesive architecture, including:

- Document Parsing: The system supports ingestion of various document formats such as PDFs and DOCX files, intelligently extracting structured text, tables, and images for downstream processing.
- Vector Embedding: Parsed documents are converted into semantic vector representations using pretrained embeddings, which enable efficient and meaningful similarity-based retrieval of relevant content.
- Hybrid Retrieval: The retrieval mechanism combines scoped search focused on explicitly referenced documents or sections and broader global search methods to ensure depth and breadth in information coverage.
- Modular Pipeline Orchestration: A multi-agent orchestration framework governs the workflow, employing conditional logic and looping to dynamically select appropriate sub-tasks based on query context and intermediate results.
- Answer Synthesis: Retrieved information is synthesized into coherent, context-aware answers by leveraging discourse context and memory of prior interactions, supporting personalized and traceable responses.

The system's design enables scalable, robust question answering grounded in heterogeneous, multi-format document collections while retaining strong provenance and user-aware memory capabilities that enhance trustworthiness and user experience.

Problem Definition

In complex knowledge domains such as regulatory, scientific, or technical fields, users frequently require precise answers grounded in extensive, diverse document corpora that include dense text, tables, and figures. Traditional question answering systems often face challenges including:

- Limited processing of multi-format documents, restricting accessibility to rich knowledge sources.

- Retrieval methods that either focus narrowly and miss broader context or search broadly but deliver noisy, less relevant results.
- Monolithic "retrieve-and-generate" architectures that do not incorporate intermediate reasoning or modular query understanding, limiting answer accuracy and explainability.
- Lack of contextual memory and user profiling, reducing personalization and continuity across interactions.

The problem we aim to solve is how to build an intelligent, scalable question answering pipeline that overcomes these limitations by unifying advanced document understanding, hybrid and hierarchical retrieval, multi-agent modular processing, and adaptive answer synthesis with memory integration.

Our Approach

To address these challenges and gaps in existing RAG systems, our project develops an Agentic RAG pipeline—a unified, modular framework that integrates multiple RAG architectures and enhanced reasoning components. Rather than stopping at the baseline "retrieve + generate" model, our approach orchestrates a sequence of specialized agents, each responsible for targeted reasoning and processing steps prior to final answer formulation. Key characteristics of our method include:

- Hybrid Retrieval Modes: Switching between scoped retrieval tightly bound to explicit query references and global retrieval to broaden search when needed, ensuring both precision and recall.
- Multi-Agent Orchestration: A state-graph driven controller dynamically routes query processing through modules such as reranking, clause location, follow-up querying, and synthesis, enabling conditional looping and adaptive workflows.
- Granular Contextual Reasoning: Agents perform fine-grained tasks like clause extraction, requirement detection, and standard citation identification, improving interpretability and traceability of answers.
- Memory and Personalization: User interaction history is integrated to provide personalized context for disambiguation, pronoun resolution, and continuity, enriching answer relevance.
- Multi-Modal Document Processing: Support for embedding and interpreting diverse document elements—text, tables, images—allows rich knowledge integration beyond plain text.

This systematic agentic design fosters a scalable and extensible platform facilitating trustworthy, explainable AI assistants specialized for complex, document-intensive knowledge domains.

2. Background

Retrieval-Augmented Generation (RAG)

Retrieval-Augmented Generation (RAG) is an advanced approach that combines the generative capabilities of large language models (LLMs) with the precision of external document retrieval systems to produce answers grounded in factual content. Instead of relying solely on the language model's parameters, which represent a fixed knowledge base, RAG dynamically fetches relevant segments—often called “document chunks” or “passages”—from external corpora or databases at query time. These retrieved chunks supplement the LLM’s context, allowing it to generate responses that are not only fluent and coherent but also verifiably linked to source documents. This technique significantly enhances the model’s accuracy and trustworthiness, particularly for knowledge-intensive domains where up-to-date or specialized information is critical. Effective retrieval of relevant document chunks is central to RAG’s success, as the quality and relevance of retrieved content directly influence the factual grounding of the generated answers.

Motivations for Agentic RAG

While traditional RAG frameworks offer a powerful mechanism for retrieval-informed generation, they tend to treat retrieval and generation as fairly monolithic steps with limited modularity or flexibility. The pipeline in question advances this paradigm by embedding RAG within an agentic architecture composed of modular processing nodes. These nodes are interconnected through a dynamically configurable directed graph called LangGraph, facilitating complex workflows and conditional routing during query handling. This agentic design enables the system to perform specialized tasks beyond simple retrieval and generation, including:

- Clause Localization: Precisely identifying and extracting relevant sections, clauses, or subsections within structured documents to improve answer specificity and provenance.
- Requirements Extraction: Automatically detecting and highlighting normative sentences or requirements (e.g., "shall," "must" clauses) that are often critical in regulatory, legal, or technical documents.
- Dynamic Reranking: Enhancing retrieval precision by re-assessing and prioritizing retrieved chunks based on semantic similarity, metadata, and contextual cues, ensuring that the most pertinent information is surfaced first.
- Persistent Chat Memory: Maintaining conversational context and historical interaction data allows the system to perform pronoun and referent resolution across turns, resulting in more coherent and context-aware multi-turn dialogues.

- Extensible Fallback to External Web Search: When internal knowledge falls short, the pipeline can gracefully integrate external web search results to broaden its information scope, enhancing robustness and reducing unanswered queries.

These capabilities collectively empower the system to act as an intelligent agent with nuanced control over the retrieval and generation workflow, offering improved accuracy, interpretability, and extensibility compared to classical RAG implementations. The modularity also facilitates easy integration of new features or updates, making the pipeline adaptable to diverse domains and requirements.

3. High-Level Architecture

Pipeline Overview

The system is built as a flexible and modular pipeline represented by a directed graph of discrete processing nodes. Each node is implemented as an independent Python function—such as n_retrieve, n_rerank, n_judge, and others—that performs a specific task on the query processing workflow. These nodes collaboratively update a shared ChatState dictionary, which holds the current state of the interaction. This dictionary includes the user's query, retrieved document sets, any web search results, generated answer text, and diagnostic traces to support debugging and transparency. By structuring the pipeline as a graph, the system enables dynamic control flow, where multiple nodes can be invoked sequentially or conditionally based on the evolving conversation context or retrieval outcomes.

Modular Node Design

Each node in the graph is dedicated to a clearly defined responsibility, allowing easy extension, customization, and maintenance:

- Retrieval (n_retrieve): Searches the internal document collections to fetch relevant chunks addressing the input question, using both semantic vector search and keyword matching.
- Reranking (n_rerank): Applies a smarter scoring mechanism utilizing semantic similarity as well as metadata information (such as section headings and page type) to reorder retrieved documents by relevance.
- Judging Retrieval Quality (n_judge): Determines if the retrieved documents sufficiently address the user query or if broader retrieval is needed. It incorporates logic to identify inherited conversation scope and handles thresholds to decide if further action is required.
- Web Search Fallback (n_web): Invoked when internal document retrieval is insufficient, this node interfaces with external web search APIs to supplement information, ensuring broader coverage.
- Clause Location (n_clause): Pinpoints exact document sections or clauses relevant to the query, enhancing answer precision especially in structured regulatory or technical documents.
- Answer Synthesis (n_synth): Generates the final answer text by aggregating retrieved content, optionally integrating chat memory context, and composing a fluent response grounded in the available data.

- Requirement Extraction (n_req): Extracts normative statements or requirements from retrieved documents, highlighting sentences containing terms like “shall,” “must,” or similar obligation indicators.
- Citations (n_cite): Attaches structured source citations to the generated answers, improving traceability and user trust.
- Topic Suggestion (n_suggest): Based on the retrieved content and conversation history, proposes relevant subtopics and follow-up questions to guide further exploration.
- Query Gap Logging (n_gap): Logs queries for which retrieval or answering was inadequate, assisting in system monitoring and improvement.

The graph’s edges define both linear progression through nodes and conditional branching, allowing flexible routing depending on intermediate results (e.g., if the judge node concludes retrieval is sufficient it proceeds to clause location, else it may trigger web search or broadened retrieval).

Agentic Capabilities
Several advanced agentic features support complex and adaptive query handling:

- Decomposition: The system can break down multi-part or compound user queries into independent, standalone questions. It also performs pronoun and referent expansion based on conversational context to ensure that each sub-question is clear and self-contained, which improves retrieval and answer generation accuracy.
- Conditional Routing: A central decision point, the judge node (n_judge), assesses retrieval quality and determines the next step, whether to continue processing, widen the search scope, or fallback to external sources.
- Fallback Mechanisms: When internal retrieval is insufficient, specialized nodes such as n_web integrate external web search results to ensure the system can provide useful information in a wider range of scenarios.
- Persistent Memory: The pipeline stores past question-and-answer pairs continuously in a persistent chat memory. This memory facilitates enhanced context for multi-turn conversations, allowing for pronoun resolution, follow-up question interpretation, and maintaining conversation continuity.

Together, these design aspects enable the system to operate as an intelligent, modular agent capable of managing complex interactions, dynamically adapting to the quality and scope of available information, and continuously enriching its responses through memory and fallback sources.

4. Data Ingestion and Processing

Document Parsing

The system supports ingestion of documents primarily in two formats: PDF and DOCX/DOC files, each parsed with tailored strategies to accurately extract and structure content.

1. PDF Parsing:
   PDF documents are processed using a combination of layout heuristics and content analysis. The parser leverages font properties (such as font size and boldness) and sophisticated regex patterns to detect and classify headings. It also intelligently identifies and excludes non-content elements like headers, footers, and any text overlapping detected tables to avoid data duplication and confusion. Structural elements such as tables are separately identified and extracted. The parser uses page bands to approximate content areas and exclude peripheral elements. Additionally, it incorporates a mechanism to discover table of contents pages and builds a whitelist of expected headings for more precise section detection.
2. DOCX Parsing:
   DOCX and DOC files are parsed using a semantic and structural approach that respects the document's inherent XML-based format. Paragraphs are segmented and analyzed to detect headings by examining styles (e.g., "Heading 1", "Heading 2") and leading numbering patterns within the text. Tables are extracted and structured into column and row arrays. For embedded images, the system uses integrated GPT-powered vision capabilities to classify content as either code snippets, tables, or plain images. Page breaks are detected through explicit markers and section properties embedded in the document, allowing for reasonable pagination even in documents without explicit page numbers.

Chunks and Metadata Extraction

Once parsed, the document content is segmented into manageable chunks optimized for retrieval and processing:

1. Chunk Segmentation:
   Text, code, tables, and images are broken down into discrete chunks to facilitate efficient indexing and searching. For text and code, chunks are split based on token limits while preserving semantic coherence. Tables are converted into structured JSON with clearly defined columns and rows.

2. Metadata Attachment:
   Each chunk is richly annotated with metadata to provide context and enhance retrieval accuracy. Metadata includes:
   - Document name and type (PDF, DOCX)
   - Page index and human-readable page labels derived from page numbering schemes
   - Section hierarchy captured through heading detection—each chunk is tagged with its section title, numerical heading, and hierarchical level, enabling precise navigation within the document structure
   - Modality tag specifying the chunk content type (e.g., text, code, table, image)
   - Timestamps indicating ingestion time for freshness tracking

3. Heading Fusion and Hierarchy Tracking:
   The system intelligently fuses heading numbering with titles (e.g., "3.2.1 Section Title") to preserve logical numbering styles present in source documents. It maintains a section stack to represent nesting and hierarchy levels, which supports building a structured outline of the document in memory. This hierarchical tracking supports advanced features like scoped query retrieval within specific document subsections.

5. Vector Store and Embedding Strategy

Chroma Vector Store

The system manages embeddings and document chunks using Chroma vector stores, which organize the data into distinct collections optimized for different use cases:

- The docs_and_headings collection holds embeddings for both document content chunks and heading-level summaries. This facilitates precise semantic search and retrieval by leveraging the document's hierarchical structure.
- The chat_memory collection stores embeddings related to user interactions and chat history, enabling efficient recall of past conversation context for improved conversational relevance.

These vector stores provide persistent storage, ensuring that embeddings and indexed chunks survive restarts and can be efficiently queried over time. They support high-performance similarity searches by leveraging approximate nearest neighbor indexing internally.

Embedding Models Usage

The embedding pipeline relies on BAAI's "bge-base-en-v1.5" model, a transformer-based embedding model that produces normalized vector embeddings. Key details include:

- Chunk and Document Level Embeddings: Input text—whether a chunk of text, code snippet, table content, or image caption—is converted into dense vector representations. Both granular chunks and aggregated document summaries receive embeddings to support layered retrieval strategies.
- Normalization: Embeddings are normalized (unit vectors) to facilitate consistent and stable cosine similarity calculations, which measure semantic closeness between query and document vectors.
- Integration via sentence-transformers: The system uses the popular sentence-transformers library interface to encode text, allowing seamless generation and comparison of embeddings. Cosine similarity is computed efficiently using matrix operations in the library (util.cos_sim), enabling rapid ranking of relevant documents or chunks against a user query.

Workflow Overview

1) Initialization and Caching: The embedding model instance is cached globally to avoid repeated loading and to optimize performance.

2) Ingestion and Indexing: During the initial ingestion or addition of new documents, extracted chunks and summaries are embedded and stored in the Chroma collections with metadata.
3) Retrieval: Query text is embedded on-the-fly using the same model, then similarity search is performed against the stored embeddings. The results are returned ranked by cosine similarity scores.
4) Multi-level Search: Retrieval operates on combined collections, allowing hybrid approaches that leverage both chunk-level and heading-level embeddings for improved precision and recall.

By using **BAAI's high-quality** normalized embeddings stored in persistent Chroma vector databases and leveraging cosine similarity ranking from sentence-transformers, the system achieves efficient, scalable, and semantically rich document retrieval tailored for complex multi-format documents and conversational memory contexts.

6. Retrieval Mechanics and Hybrid Systems

Chunk Retriever

Semantic search is primarily enabled through the chunk retriever, which performs similarity searches on chunk-level embeddings stored in the vector store. When a query is issued, the system encodes it into an embedding using the same embedding model and retrieves document chunks ranked by cosine similarity. This allows the retrieval of semantically relevant chunks, even if they do not share exact keywords with the query, enhancing the system's understanding and relevant content discovery.

BM25 Retriever

To complement semantic retrieval, the system also employs a BM25 retriever, which is a classic keyword-based retrieval mechanism. BM25 serves to improve recall by capturing documents and chunks that match query keywords more directly but may be semantically less nuanced. This approach helps cover cases where keyword presence is crucial and ensures that documents relevant by literal term matching are not missed.

The BM25 retriever Is built from the collection of all documents and supports retrieval with a parameter to fetch the top-k relevant documents by keyword ranking.

Hybrid Retrieval Flow

Hybrid retrieval is orchestrated within the n_retrieve node, which unifies both semantic and keyword-based search results while applying additional scoping and filtering for precision:

- Combination of Results: The node queries the chunk retriever and BM25 retriever separately, then merges their results to leverage complementary strengths of semantic and lexical matching.
- Scoped Document Filtering: If the query explicitly mentions a regulatory document code or name (detected via pattern matching and normalization), the retrieval scope is restricted strictly to that document's content. This scoped search ensures precise and contextually relevant results when users reference specific regulatory documents.
- Deduplication: Retrieved results from the different retrieval methods are deduplicated by their document name, section, and chunk identity, preventing redundant documents in the final combined set.
- Result Capping: To maintain performance and manageable result sizes, the system caps the number of retrieved documents at 100 entries. This cap helps balance completeness

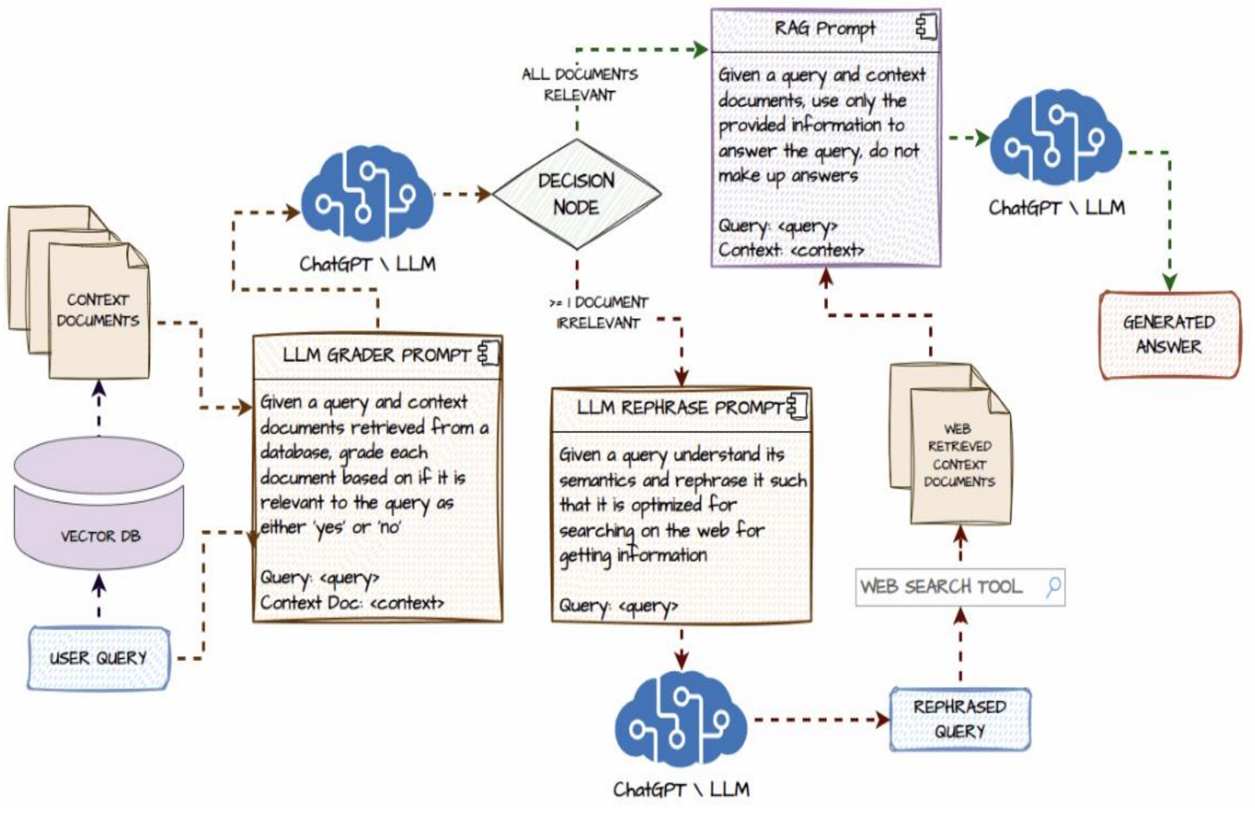
with efficiency, especially when queries are broad or multiple retrieval modalities return overlapping results.

- Fallback and Trace: In the absence of scoped targets or if broader search is needed, the node optionally performs a broadened global search across collections, combining and ranking results accordingly while recording trace logs for diagnostic and analytics purposes.

This hybrid retrieval approach allows the system to flexibly respond to varying query intents—whether seeking broad semantic matches, keyword-specific hits, or targeted document sections—optimizing both recall and precision in complex regulatory information retrieval contexts.

7. Agentic Graph Pipeline (LangGraph)

Agentic Graph working:


CONTEXT DOCUMENTS
VECTOR DB
USER QUERY
ChatGPT \ LLM
LLM GRADER PROMPT
Given a query and context documents retrieved from a database, grade each document based on if it is relevant to the query as either 'yes' or 'no'
Query: <query>
Context Doc: <context>
DECISION NODE
ALL DOCUMENTS RELEVANT
>= 1 DOCUMENT IRRELEVANT
RAG Prompt
Given a query and context documents, use only the provided information to answer the query, do not make up answers
Query: <query>
Context: <context>
ChatGPT \ LLM
GENERATED ANSWER
LLM REPHRASE PROMPT
Given a query understand its semantics and rephrase it such that it is optimized for searching on the web for getting information
Query: <query>
ChatGPT \ LLM
REPHRASED QUERY
WEB SEARCH TOOL
WEB RETREVED CONTEXT DOCUMENTS

Node Definitions and Routing

The Agentic Graph Pipeline, implemented as LangGraph, is a modular and extensible orchestration engine where each node represents a distinct processing step in a question-answering workflow over regulatory documents. Its core nodes and functions include:

- n_retrieve: Executes a hybrid retrieval strategy combining vector embeddings and BM25 keyword matches, applying strict scope filtering when specific documents are mentioned explicitly.
- n_rerank: Reorders retrieved documents by computing embedding similarity enhanced with metadata-aware boosts such as heading relevance, word counts, and section types to prioritize the most relevant chunks.
- n_judge: Evaluates whether the retrieved context is sufficient for answering the user query or if a broader search is necessary. It considers explicit document mentions, inheritance of scope from previous queries, and similarity thresholds to make this decision.
- n_web: Provides a fallback to web search via an external API when internal retrieval is insufficient and web search capability is enabled.
- n_clause: Identifies specific document clauses and sections relevant to the query by embedding similarity and heading term overlap.
- n_follow: Determines if a follow-up query is needed because the current context is insufficient or ambiguous.
- n_synth: Synthesizes a final, concise answer by combining retrieved text chunks, query context, and chat memory via prompt engineering with the language model.
- n_req: Extracts normative "shall/must/should" requirements from the retrieved text fragments.
- n_ext: Detects and attaches references to external standards (such as IAEA, CSA, ISO) in the answer.
- n_cite: Formats and appends valid source citations to the answer for traceability and trust.
- n_suggest: Generates suggestions for related subtopics and follow-up questions based on the current context.
- n_gap: Logs queries that lack sufficient retrieval context to support ongoing system improvement.

Routing Strategy

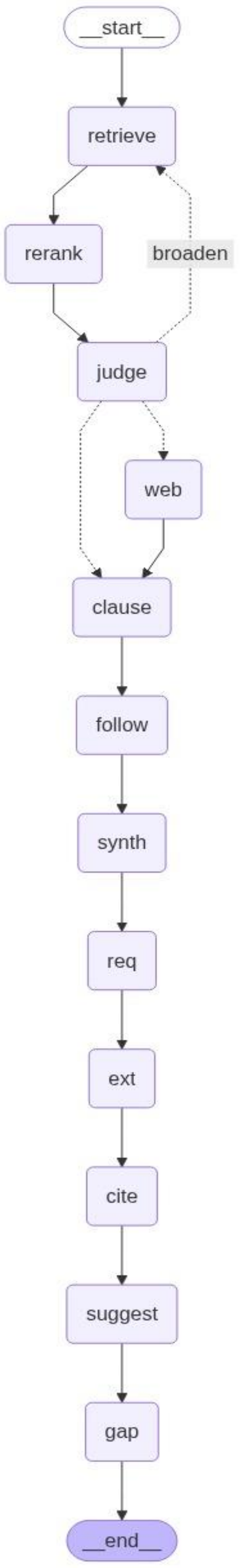
- Central to the pipeline is the _route_after_judge function that routes the flow based on the decision by n_judge.

- If n_judge signals that more data is needed (broaden_needed is True), the pipeline loops back to the n_retrieve node to perform a broader retrieval.
- If the retrieved context is sufficient (enough is True), processing advances to n_clause for clause extraction and further answer refinement.
- Otherwise, if internal data is insufficient, the pipeline moves to n_web for web search fallback.

Orchestration and Modularity

- The pipeline is implemented via a StateGraph object, which composes nodes and edges with conditional transitions, supporting modular node additions and clear interface definitions.
- Linear and conditional paths define strict but flexible progression through retrieval, reranking, judgement, clause resolution, synthesis, and supplementary processes like citation and suggestion generation.

Agentic Graph Pipeline Workflow Diagram


__start__
retrieve
rerank
broaden
judge
web
clause
follow
synth
req
ext
cite
suggest
gap
__end__

8. Memory and Chat Context Handling in the Agentic Graph Pipeline

Persistent User Memory Storage

- The system persistently stores all user and assistant utterances, each annotated with metadata including the user ID, chat session ID (chat_id), timestamps, and references to relevant document chunk IDs (doc_ids).
- This memory is stored in a dedicated vector store (memory_vs) that enables semantic similarity search over past interactions.
- The storage function (write_memory) takes the utterance role (user or assistant), the text, document IDs it relates to, user, chat ID, and an optional timestamp, and saves them into the memory vector store for later retrieval.

Semantic Memory Retrieval

- Upon new queries, the system retrieves the top k most semantically similar past conversation turns from the stored memory for the same user and chat session.
- This retrieval, handled by the recall_memory function, uses vector similarity search over the memory vector store, filtered by user and chat_id metadata to limit to relevant conversations.
- Retrieved past turns may include both the text of the utterance and its metadata (if requested).

Memory Injection into Prompts

- Before generating an answer, the pipeline injects recent past interactions (up to k closest turns) as plain text blocks into the prompt context for the language model.
- The injection is done in a user/assistant conversational format, enabling the model to resolve ambiguous references such as pronouns ("it," "they," "those") by providing relevant prior context.
- This context management happens in the memory_context function, which formats the retrieved memory turns for seamless prompt integration.

Enhancing Multi-Turn Coherence

- By leveraging semantic similarity between the new query and stored past chat turns, memory is used to:
- Maintain user-specific context across multiple question-answer rounds.
- Resolve pronouns and anaphora, ensuring queries are expanded or rewritten with clearer referents internally.
- Potentially seed or constrain retrieval and synthesis steps with relevant past information, improving responsiveness and answer accuracy.

Integration in Query Processing Pipeline

- Prior to query decomposition and execution, the system checks if memory usage is enabled and pulls the relevant memory data for the current user and chat.
- It identifies the last document or scope targeted (last_target) and any initial retrieval seed query (retrieval_seed) from previous interactions to better frame the current query.
- Memory feedback loops help update the current context after each answer, persisting both question and response back to memory for the ongoing conversation.

9. Synthesis and Answer Generation in the Agentic Graph Pipeline

Contextual Prompt Engineering for Synthesis (n_synth)

- The synthesis node (n_synth) generates the final answer using an LLM based on retrieved documents and chat memory context.
- Prompts include relevant retrieved context and user questions to produce focused and coherent responses.

Enumeration Fast-Paths: Handling List Queries

- Detects enumeration or list-style questions (e.g., “What are the 3 pillars of X?”) via pattern matching.
- Extracts list items directly from the identified document section text.
- Returns answers as enumerated lists with strong provenance referencing exact document sections.

Clause-Location Fast-Paths: Exact Section Returns with Provenance

- Returns exact section or clause headings with detailed provenance when explicitly requested.
- Provenance includes document name, section number, title, and page number for traceability.
- Stores question and answer in chat memory if enabled.

Web Fallback Synthesis

- Integrates web search results as supplementary context when local retrieval is insufficient.
- Enables synthesis across both retrieved documents and current web data.
- Triggered when confidence in local retrieval is low or additional information is desired.

Final Fallback: Synthesizing Across All Retrieved Data

- Concatenates all relevant retrieved document chunks as context for the LLM.
- Generates a comprehensive, concise answer covering available information.
- Ensures an answer is produced even with sparse or scattered data.

Final Answer Fusing for Multi-Part Questions

- Decomposes multi-part user queries into standalone sub-questions.

- Processes each independently to produce partial answers.
- Combines partial answers in a final fusion step via LLM into a cohesive, fluent response.

This summary retains all key points of the synthesis and answer generation workflow, omitting intricate code details and auxiliary explanations for clarity and brevity.

10. Results

Retrieval Quality

The system employs a multi-layered retrieval architecture designed to enhance both precision and coverage. Initially, a broad retrieval gathers relevant document chunks combining semantic vector search and BM25 keyword matching. This pool is then refined through a reranking stage that incorporates not only text similarity via embeddings but also metadata signals such as document section relevance, length heuristics, and heading-title semantic alignment. Additionally, explicit user context and document scope are leveraged to further prune and focus retrieval results. While empirical evaluation metrics are not directly derivable from the code alone, this comprehensive retrieval and reranking strategy optimizes for returning highly relevant, contextually appropriate chunks, improving answer accuracy and meaningfulness.

Sample Result Screenshots

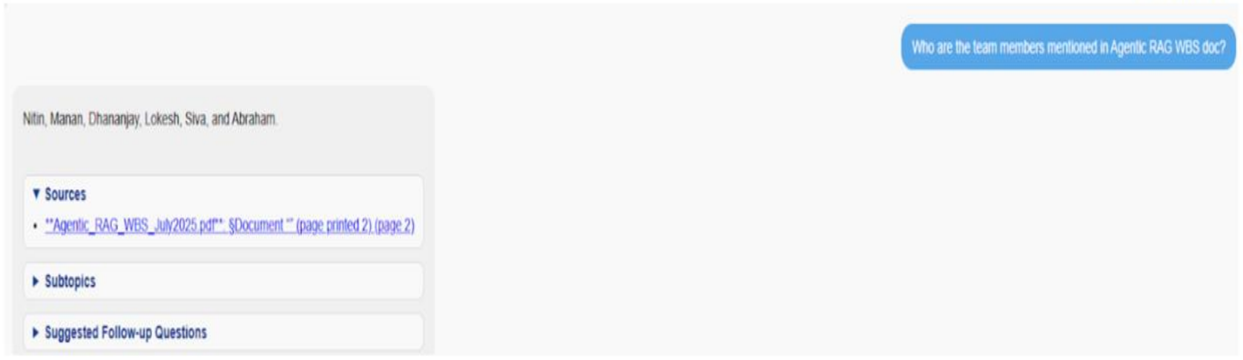
At runtime, detailed logs capture key retrieval and synthesis outputs which can be used to generate illustrative screenshots demonstrating the system's performance:

- Retrieved Document Chunks: The system produces a ranked list of retrieved document chunks, each annotated with associated similarity scores reflecting relevance to the query. Accompanying metadata such as document names, section headings, and detailed retrieval traces are logged, offering clear insight into which parts of documents contributed to the response formulation.

- Synthesized Answers: Answers synthesized by the language model dynamically integrate and fuse information extracted from multiple retrieved chunks. The system's output includes coherent, context-aware responses that handle complex, multi-part queries by blending relevant content with user interaction context

- Citation and Requirement Extraction: Normative requirement statements containing keywords like "shall" and "must" are detected and extracted from source documents. These extracted requirements are linked to their exact source sections with precise citations, enabling improved traceability and confidence in the answers.

Demo


Who are the team members mentioned in Agentic RAG WBS doc?

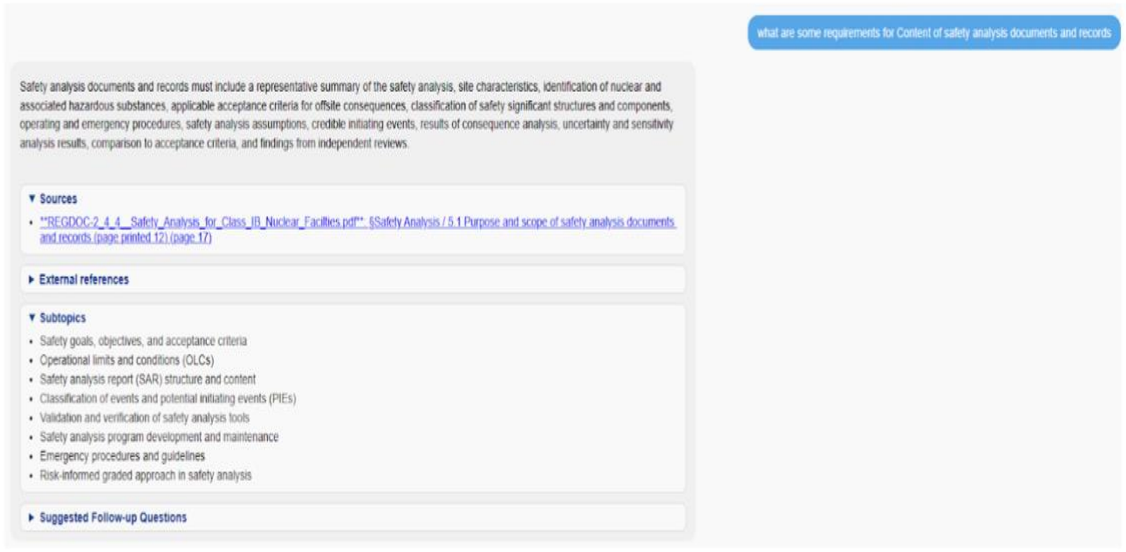
Nitin, Manan, Dhananjay, Lokesh, Siva, and Abraham.

▼ Sources

- **Agentic_RAG_WBS_July2025.pdf**: §Document "" (page printed 2) (page 2)

▶ Subtopics

▶ Suggested Follow-up Questions


what are some requirements for Content of safety analysis documents and records

Safety analysis documents and records must include a representative summary of the safety analysis, site characteristics, identification of nuclear and associated hazardous substances, applicable acceptance criteria for offsite consequences, classification of safety significant structures and components, operating and emergency procedures, safety analysis assumptions, credible initiating events, results of consequence analysis, uncertainty and sensitivity analysis results, comparison to acceptance criteria, and findings from independent reviews.

▼ Sources

- **REGDOC-2_4_4__Safety_Analysis_for_Class_IB_Nuclear_Facilties.pdf**: §Safety Analysis / 5.1 Purpose and scope of safety analysis documents and records (page printed 12) (page 17)

▶ External references

▼ Subtopics

- Safety goals, objectives, and acceptance criteria
- Operational limits and conditions (OLCs)
- Safety analysis report (SAR) structure and content
- Classification of events and potential initiating events (PIEs)
- Validation and verification of safety analysis tools
- Safety analysis program development and maintenance
- Emergency procedures and guidelines
- Risk-informed graded approach in safety analysis

▶ Suggested Follow-up Questions

11. Conclusions, Discussion and Future Work

The presented agentic RAG pipeline code constitutes a comprehensive and sophisticated framework for document-grounded question answering. By tightly integrating hybrid retrieval strategies, advanced synthesis techniques, memory utilization, and multi-agent orchestrated processing, it establishes a scalable and extensible platform. This platform is well-suited for research and practical deployment of reliable, trustworthy AI assistants, particularly within regulatory and scientific domains where accuracy, traceability, and context-awareness are critical.

The code architecture reflects significant advancements beyond traditional Retrieval-Augmented Generation (RAG) systems through several innovative features:

- Robust Hybrid Retrieval: The system incorporates both scoped retrieval, focusing on explicitly referenced documents or sections, and global retrieval, which broadens the search space when needed. This dual-mode approach improves precision while maintaining comprehensive coverage.
- Multi-Agent Orchestration with Conditional Looping: A state-graph manages complex workflows with conditional transitions, enabling the system to loop back to retrieve more information if initial results are insufficient, or to proceed through stages like reranking, clause location, and answer synthesis dynamically.
- Dynamic Answer Synthesis Guided by Discourse Context and Memory: The synthesis module integrates context from retrieved documents with user-specific memory of past interactions, allowing more coherent, context-aware, and personalized responses.

Potential avenues for future enhancements include:

- Enhanced Multi-Modal Document Ingestion: Extending capabilities to more effectively ingest and interpret images, charts, and other non-textual data embedded within documents would enrich the information base for retrieval and synthesis.
- Integration of Knowledge Graph Reasoning: Adding reasoning over structured knowledge graphs could support more nuanced inferencing, enabling the system to answer complex queries involving relationships and constraints beyond text matching.
- More Granular User Profiling in Memory: Developing finer-grained user models stored in memory could improve personalization, tailoring retrieval, and responses more closely to user preferences and needs over extended interactions.