IKMS Multi-Agent RAG Extension Features

Project Overview

This document outlines **5 enhancement features** for Sachith's existing Multi-Agent RAG (Retrieval-Augmented Generation) project. Students will select **one feature** to implement, which includes:

- Building upon the existing IKMS codebase
- Implementing a complete user interface
- Deploying the enhanced application

Existing Project Architecture

The current project implements a **multi-agent RAG pipeline** with the following technology stack:

Technical Framework & Implementation

- **Framework Foundation**: The system is built with LangChain v1.0
- **Continuation Options**: Students can continue with the existing framework
- **Learning Recommendation**: Research LangChain v1.0 concepts, including new middleware features for summarization

Core Technologies

- **LangChain v1**: `ChatOpenAI`, tools, retriever abstractions
- **LangGraph**: Linear multi-agent graph orchestration
- **Pinecone**: Vector database for document storage
- **FastAPI**: Backend API with `/qa` and `/index-pdf` endpoints

Key Components Students Should Review

📁 Essential Files to Understand

- `src/app/core/agents/agents.py` - Retrieval, Summarization, Verification agents + node functions
- `src/app/core/agents/graph.py` - LangGraph StateGraph wiring the linear QA flow
- `src/app/core/agents/state.py` - QAState schema definition
- `src/app/core/agents/tools.py` - retrieval_tool for Pinecone queries
- `src/app/core/retrieval/vector_store.py` - Pinecone setup and PDF indexing
- `src/app/core/retrieval/serialization.py` - Chunk-to-context-string conversion
- `src/app/services/qa_service.py` - Service facade over LangGraph QA flow
- `src/app/api.py` - FastAPI endpoint definitions

Feature Selection Criteria

Each feature is designed to:

- ✅ Focus on **agentic behavior** and **multi-agent coordination**
- ✅ Emphasize **retrieval and vector store** usage
- ✅ Work as a **self-contained mini-project**
- ✅ Build directly on existing code patterns
- ✅ Require UI implementation and deployment

> ⚠️ **Important**: Frontend development is **required** for all features. While the core work happens at the API/agent layer, students must create a user interface to demonstrate their implementation.

Feature Options

Feature 1: Query Planning & Decomposition Agent

Concept: Add an intelligent **Query Planner Agent** that analyzes complex questions and creates a structured search strategy before retrieval begins.

Current Limitation

The `retrieval_node` sends the raw user question directly to the retrieval agent as a single message. The system performs only one retrieval call and may miss important aspects of complex, multi-part questions.

Example Problem:

- Question: "What are the advantages of vector databases compared to traditional databases, and how do they handle scalability?"
- Current behavior: Retrieves chunks matching the whole question, potentially missing specific details about advantages OR scalability

What You'll Build

1. Enhanced State Schema (`state.py`)

Add new fields to track planning:

```
plan: str | None  # Natural language search strategy
sub_questions: list[str] | None  # Decomposed questions
```

2. Planning Agent (`prompts.py`, `agents.py`)

Create a new agent with a system prompt that:

- Rephrases ambiguous questions
- Identifies key entities, time ranges, or topics
- Decomposes complex questions into focused sub-questions
- Outputs a structured search plan

Example Planning Output:

```
Original Question: "What are the advantages of vector databases compared to traditional databases, and how do they handle scalability?"
```

```
Plan:
1. Search for advantages of vector databases
2. Search for comparison with traditional databases
3. Search for scalability mechanisms in vector databases

Sub-questions:
- "vector database advantages benefits"
- "vector database vs relational database comparison"
- "vector database scalability architecture"
```

3. Graph Node Implementation (`graph.py`)

Insert planning node into the pipeline:

```
START → planning → retrieval → summarization → verification → END
```

The planning node:

- Reads `state["question"]`
- Invokes the planning agent
- Extracts and stores the plan in state

4. Enhanced Retrieval

Update `retrieval_node` to use the plan:

- Include both original question AND plan in retrieval messages
- Optionally enable multiple retrieval tool calls for each sub-question
- Aggregate results from all sub-queries

Acceptance Criteria

✅ Complex questions trigger a visible planning step in logs

✅ Retrieval behavior changes based on generated plan (more relevant/diverse chunks)

✅ Downstream agents (summarization, verification) continue working without modification

✅ API exposes the generated plan in response

UI Implementation Ideas

- Display the generated search plan above the final answer
- Show which sub-questions were created
- Visualize the planning → retrieval → answer flow
- Add a toggle to enable/disable query planning

Feature 2: Multi-Call Retrieval Tool with Message Organization

Concept: Transform the Retrieval Agent into a sophisticated **tool-using agent** that can make multiple strategic retrieval calls and properly organize all retrieved information.

Current Limitation

The `retrieval_tool` supports multiple calls, but `retrieval_node` only keeps the **last ToolMessage** content. Earlier tool calls are silently discarded, and the artifact information (document metadata) is lost.

Example Problem:

- Agent makes 3 retrieval calls with different query variations
- Only the result from call #3 is passed to summarization
- Potentially valuable context from calls #1 and #2 is lost

What You'll Build

1. Comprehensive Message Aggregation (`retrieval_node`)

Parse ALL `ToolMessage` instances from the message list:

```
retrieval_traces = []
for i, tool_msg in enumerate(tool_messages):
    retrieval_traces.append({
        "call_number": i + 1,
        "query_variant": extract_query(tool_msg),
        "context_block": tool_msg.content,
        "artifacts": extract_artifacts(tool_msg)
    })
```

2. Enhanced State Fields (`state.py`)

Add transparency fields:

```
retrieval_traces: str | None  # Human-readable log
raw_context_blocks: list[str] | None  # Individual contexts
```

Example Retrieval Trace:

```
Retrieval Call 1:
Query: "vector database indexing"
Chunks Retrieved: 5
Sources: Pages 3, 7, 12

Retrieval Call 2:
Query: "vector database query performance"
Chunks Retrieved: 4
Sources: Pages 8, 15, 16

Retrieval Call 3:
Query: "vector database approximate nearest neighbor"
Chunks Retrieved: 6
Sources: Pages 4, 9, 11
```

3. Structured Context Assembly

Instead of concatenating chunks, create organized context:

```
=== RETRIEVAL CALL 1 (query: "vector database indexing") ===

Chunk 1 (page 3): [content]
Chunk 2 (page 7): [content]

=== RETRIEVAL CALL 2 (query: "vector database query performance") ===

Chunk 3 (page 8): [content]
Chunk 4 (page 15): [content]
```

4. Message Organization Learning

Students experiment with:

- Different context structuring approaches
- Minimal vs. comprehensive context schemas
- Impact on downstream agent performance

Acceptance Criteria

✅ Logs show **multiple ToolMessages** being captured

✅ `QAState` includes `retrieval_traces` documenting all calls

✅ Downstream agents receive structured, organized context

✅ No retrieval information is silently discarded

UI Implementation Ideas

- Debug panel showing all retrieval calls
- Expandable sections for each retrieval attempt
- Metadata display (query used, chunks found, sources)
- Visual indication of which chunks contributed to the final answer

---

Feature 3: Context Critic & Reranker Agent

Concept: Insert an intelligent **Context Critic Agent** between retrieval and summarization that filters, ranks, and annotates chunks to ensure only highly relevant evidence reaches the answer generation stage.

Current Limitation

All retrieved chunks are passed directly to summarization without quality assessment. This means:

- Irrelevant or low-quality chunks consume token budget
- Summarization agent must process noise alongside signal
- Similar terms from unrelated sections create confusion

Example Problem:

- Question: "How do vector databases handle concurrent writes?"
- Retrieved chunks include:
  - ✅ Page 14: Detailed section on concurrent write mechanisms
  - ✅ Page 19: Write performance benchmarks
  - ❌ Page 3: General database concurrency (not vector-specific)
  - ❌ Page 8: "write" mentioned in different context (write-ahead logs)
  - ❌ Page 22: Concurrent reads (not writes)
- Currently, all 5 chunks pass to summarization

What You'll Build

1. Enhanced Retrieval State (`retrieval_node`)

Store both processed and raw context:

```
state["context"] = formatted_context_string
state["raw_docs"] = original_document_list
state["raw_context_blocks"] = individual_chunk_strings
```

2. Context Critic Agent (`prompts.py`, `agents.py`)

Create an agent with a system prompt to:

- Analyze each chunk's relevance to the question
- Assign relevance scores (Highly Relevant / Marginal / Irrelevant)
- Provide brief rationales for each judgment
- Filter or de-prioritize low-relevance chunks
- Reorder chunks by relevance

Example Critic Output:

```
Chunk Analysis for Question: "How do vector databases handle concurrent wr
ites?"

✅ Chunk 1 (Page 14) - HIGHLY RELEVANT
Rationale: Directly discusses concurrent write mechanisms and locking strate
gies

✅ Chunk 2 (Page 19) - HIGHLY RELEVANT
Rationale: Contains performance data for concurrent write scenarios

⚠️ Chunk 3 (Page 3) - MARGINAL
Rationale: Discusses general database concurrency but lacks vector database
specifics

❌ Chunk 4 (Page 8) - IRRELEVANT
Rationale: "Write" refers to write-ahead logs, not concurrent write operations

❌ Chunk 5 (Page 22) - IRRELEVANT
Rationale: Discusses concurrent reads, not writes

Filtered Context: Keeping Chunks 1, 2; Including Chunk 3 with lower priority
```

3. Context Critic Node (`agents.py`)

Implement the new node:

```
def context_critic_node(state: QAState) → dict:
    # Read question and raw context
    # Invoke critic agent
    # Output filtered context and rationale
    return {
        "context": filtered_context,
        "context_rationale": critic_reasoning
    }
```

4. Updated Graph Flow (`graph.py`)

```
START → retrieval → context_critic → summarization → verification → END
```

Acceptance Criteria

✅ Questions with noisy retrieval results show visible filtering

✅ Summarization receives critic-filtered context, not raw retrieval output

✅ Logs expose which chunks were kept/filtered and why

✅ Answer quality improves for questions with ambiguous keywords

UI Implementation Ideas

- Side-by-side view: "Before Critic" vs "After Critic" context
- Visual indicators for chunk relevance (color coding)
- Display critic's rationale for each chunk
- Statistics: X chunks retrieved → Y chunks after filtering
- Toggle to bypass critic and compare results

Feature 4: Evidence-Aware Answers with Chunk Citations

Concept: Enhance the pipeline to generate answers with **explicit, traceable citations** back to specific chunks and pages, enabling full transparency about

evidence sources.

Current Limitation

- `serialize_chunks` labels chunks but doesn't create stable IDs
- Final answers are plain text with no source references
- Users cannot verify which parts of the answer came from which chunks
- API returns only `answer` and `context` without citation mapping

Example Problem:

```
Question: "What are the main indexing strategies in vector databases?"

Current Answer:
"Vector databases use several indexing strategies. HNSW provides fast appro
ximate search
through hierarchical graphs. LSH uses hash functions for similarity. IVF partiti
ons the
vector space into clusters."

Problem: No way to know which chunks contributed which statements!
```

What You'll Build

1. Stable Chunk Identifiers (`serialization.py`)

Generate unique, stable IDs for each chunk:

```
def serialize_chunks_with_ids(docs: list[Document]) → tuple[str, dict]:
    context_parts = []
    citation_map = {}

    for i, doc in enumerate(docs):
        chunk_id = f"C{i+1}"
        page = doc.metadata.get("page", "unknown")

        context_parts.append(f"[{chunk_id}] Chunk from page {page}:\n{doc.pa
```

```
ge_content}")

        citation_map[chunk_id] = {
            "page": page,
            "snippet": doc.page_content[:100] + "...",
            "source": doc.metadata.get("source", "unknown")
        }

    return "\n\n".join(context_parts), citation_map
```

2. Citation-Aware Agent Prompts (`prompts.py`)

Update system prompts:

```
When answering, you MUST cite your sources using the chunk IDs provided i
n the context.

Format: Include [C1], [C2], etc. immediately after statements derived from tho
se chunks.

Example:
"HNSW indexing creates hierarchical graphs for efficient search [C1]. This ap
proach
offers better recall than LSH methods [C3][C5]."

Rules:
- Only cite chunks actually present in the context
- Use multiple citations when combining information from multiple chunks
- Do not invent or guess chunk IDs
```

3. Enhanced State and API (`state.py` , `models.py` , `api.py`)

Extend state schema:

```
citations: dict[str, dict] | None  # Chunk ID → metadata mapping
```

Update API response:

```
class QAResponse(BaseModel):
    answer: str
    context: str
    citations: dict[str, dict] | None
```

Example API Response:

```
{
  "answer": "Vector databases use several indexing strategies. HNSW provide
s fast approximate search through hierarchical graphs [C1][C2]. LSH uses has
h functions for similarity [C4]. IVF partitions the vector space into clusters [C
3].",
  "citations": {
    "C1": {
      "page": 5,
      "snippet": "HNSW (Hierarchical Navigable Small World) graphs provide log
arithmic search complexity...",
      "source": "vector_db_paper.pdf"
    },
    "C2": {
      "page": 6,
      "snippet": "The hierarchical structure allows efficient approximate nearest
neighbor search...",
      "source": "vector_db_paper.pdf"
    },
    "C3": {
      "page": 8,
      "snippet": "Inverted File (IVF) indexing partitions vectors into Voronoi cell
s...",
      "source": "vector_db_paper.pdf"
    },
    "C4": {
      "page": 7,
      "snippet": "Locality-Sensitive Hashing (LSH) maps similar vectors to the s
ame hash buckets...",
```

```
    "source": "vector_db_paper.pdf"
   }
  }
 }
```

4. Verification-Aware Citations (`verification_node`)

Prompt the Verification Agent to:

- Maintain citation consistency when correcting content
- Remove citations if associated content is removed
- Add citations if introducing new information from context

Acceptance Criteria

✅ Answers include inline citations like [C1], [C2]

✅ API exposes machine-readable citation mappings

✅ Every citation corresponds to an actual retrieved chunk

✅ Citation IDs remain stable throughout the pipeline

✅ Verification step maintains citation accuracy

UI Implementation Ideas

- **Interactive citations**: Click [C1] to highlight the source chunk
- **Hover tooltips**: Show chunk snippet when hovering over citations
- **Source panel**: Display all cited chunks with page numbers
- **Citation heatmap**: Visual indication of which chunks were most cited
- **Fact-checking mode**: Click any sentence to see its evidence sources

---

Feature 5: Conversational Multi-Turn QA with Memory

Concept: Transform the single-shot QA pipeline into a **conversational assistant** that maintains context across multiple turns, using LangGraph state to remember previous questions and answers.

Current Limitation

Each question is processed independently:

- `QAState` only contains current question data
- `run_qa_flow(question)` creates fresh state every time
- No memory of previous interactions
- Follow-up questions fail without context

Example Problem:

```
Turn 1:
User: "What is HNSW indexing?"
System: "HNSW (Hierarchical Navigable Small World) is an indexing metho
d..."

Turn 2:
User: "What are its main advantages?"
System: ❌ Doesn't know what "its" refers to
        ❌ Cannot leverage previous answer
        ❌ Must re-retrieve information about HNSW
```

What You'll Build

1. Conversational State Schema (`state.py`)

Extend `QAState` with memory fields:

```
history: list[dict] | None  # Previous turns
conversation_summary: str | None  # Compressed history
session_id: str | None  # Conversation identifier
```

Example History Structure:

```
history = [
   {
      "turn": 1,
```

```
        "question": "What is HNSW indexing?",
        "answer": "HNSW (Hierarchical Navigable Small World)...",
        "context_used": ["C1", "C2", "C3"],
        "timestamp": "2025-12-26T17:30:00Z"
    },
    {
        "turn": 2,
        "question": "How does it compare to LSH?",
        "answer": "Compared to LSH, HNSW offers...",
        "context_used": ["C1", "C4", "C5"],
        "timestamp": "2025-12-26T17:31:15Z"
    }
]
```

2. History-Aware Flow Function (`graph.py`)

Modify or create new function:

```
def run_conversational_qa_flow(
    question: str,
    history: list[dict] | None = None,
    session_id: str | None = None
) -> QAState:
    initial_state = {
        "question": question,
        "history": history or [],
        "session_id": session_id or generate_session_id(),
        # ... other fields
    }
    return graph.invoke(initial_state)
```

3. Conversational Agent Prompts (`prompts.py`)

Update all agent prompts to be history-aware:

Retrieval Agent:

```
You are a retrieval agent in a conversational system.

Current Question: {question}

Conversation History:
{history}

Tasks:
1. Analyze if this is a follow-up question referencing previous turns
2. Identify what needs to be retrieved considering the conversation context
3. Use previous answers to refine your search strategy
4. Retrieve information that complements (not duplicates) previous context
```

Summarization Agent:

```
You are answering a question in an ongoing conversation.

Conversation History:
{history}

Current Question: {question}
Retrieved Context: {context}

Tasks:
1. Use conversation history to understand references ("it", "that", "the method mentioned earlier")
2. Provide answers that build on previous turns
3. Reference previous answers when relevant
4. Avoid repeating information already provided unless specifically asked
```

4. Optional: Memory Summarization Node

Add a `memory_summarizer` node after verification:

```
def memory_summarizer_node(state: QAState) → dict:
    # After every N turns or when history is long
```

```
    if len(state["history"]) > 5:
        # Compress older turns into a summary
        summary = summarize_conversation(state["history"])
        return {"conversation_summary": summary}
    return {}
```

Example Summary:

```
Conversation Summary (Turns 1-5):
User has been exploring vector database indexing methods. Discussed HNSW indexing
in detail, compared it with LSH, explored performance trade-offs, and asked a bout
implementation considerations. Key topics: HNSW structure, search complexit y,
LSH limitations, recall-latency trade-offs.
```

5. Session Management API (`api.py`)

Add session endpoints:

```
@app.post("/qa/conversation")
async def conversational_qa(
    request: ConversationalQARequest
) → ConversationalQAResponse:
    # request.session_id, request.question
    # Retrieve history for session
    # Run conversational flow
    # Update session history
    # Return answer + updated session

@app.get("/qa/session/{session_id}/history")
async def get_conversation_history(
    session_id: str
) → ConversationHistory:
    # Return all turns for this session
```

Acceptance Criteria

✅ Follow-up questions get context-aware answers

✅ System resolves references ("it", "that", "the approach mentioned earlier")

✅ History is maintained and passed through graph state

✅ API supports session management

✅ Optional: Long conversations are summarized to manage token limits

UI Implementation Ideas

- **Chat interface** with message history
- **Session management**: New conversation, continue previous session
- **History panel**: Show all previous turns in current session
- **Context highlights**: Visual indication when answer uses previous turns
- **Memory inspector**: View the conversation summary
- **Turn expansion**: Click any previous turn to see its full context and retrieved chunks

Example UI Flow:

[Session: abc-123] [New Session]

💬 You (Turn 1):
"What is HNSW indexing?"

🤖 Assistant:
"HNSW (Hierarchical Navigable Small World) is an indexing method for approximate nearest neighbor search [C1][C2]..."
[Retrieved from 5 chunks] [Used 3 chunks in answer]

💬 You (Turn 2):
"What are its advantages over LSH?"

🤖 Assistant:
"Compared to the LSH method, HNSW offers several advantages [C4]..."
[ℹ️ Using context from previous turn]
[Retrieved 4 new chunks]

Implementation Guidelines

General Requirements for All Features

Backend Requirements

- Modify LangGraph state schema appropriately
- Update or create new agent nodes
- Adjust system prompts for new behavior
- Maintain compatibility with existing pipeline
- Add appropriate logging for debugging

Frontend Requirements

- Create a user interface that demonstrates the feature and the system built
- Implement any API integrations present
- Provide visual feedback for agent operations
- Include error handling and loading states

Deployment Requirements

- Prepare deployment configuration
- Document environment variables
- Include setup instructions

- Ensure production-ready error handling

Learning Objectives

Through implementing any of these features, students will learn:

LangGraph & State Management

- How to design and extend state schemas
- How to wire nodes in a multi-agent graph
- How to manage state propagation between agents

Prompt Engineering

- Writing effective system prompts for specialized agents
- Designing prompts for tool usage
- Balancing specificity and flexibility

Message & Context Organization

- How to structure information for LLM consumption
- Trade-offs between verbosity and clarity
- Impact of context organization on answer quality

Retrieval & Vector Databases

- Advanced retrieval patterns
- Multi-call retrieval strategies
- Context filtering and ranking

Agentic Behavior

- How agents use tools iteratively
- Decision-making in multi-agent systems
- Coordinating specialized agents

Evaluation Criteria

Technical Implementation (40%)

- Correct implementation of core feature
- Proper state management in LangGraph
- Effective prompt engineering
- Code quality and organization

Functionality (30%)

- Feature works as specified
- Acceptance criteria are met
- Edge cases are handled
- Performance is acceptable

User Interface (20%)

- Clear and intuitive UI
- Effective visualization of agent behavior
- Responsive design
- Good user experience

Deployment & Documentation (10%)

- Successful deployment
- Clear setup instructions
- Code documentation
- User guide or README

Getting Started

Step 1: Choose Your Feature

Review all five features and select one based on:

- Your interest in the specific agent behavior
- Complexity level you're comfortable with
- UI ideas that excite you

Step 2: Understand the Codebase

Before implementing:

1. Run the existing IKMS project locally
2. Review the key files listed in Project Overview
3. Test the current `/qa` endpoint
4. Trace through the graph flow

Step 3: Plan Your Implementation

1. Sketch out state changes needed
2. Design new agents and prompts
3. Plan graph modifications
4. Wireframe your UI
5. List deployment requirements

Step 4: Implement Incrementally

1. Start with state schema changes
2. Implement agent logic
3. Update graph wiring
4. Test at the API level
5. Build the UI
6. Deploy and document

Step 5: Test & Refine

- Test with various question types

- Verify acceptance criteria
- Gather feedback
- Refine prompts and UI

Questions & Support

If you have questions about:

- **Feature selection**: Discuss which feature aligns with your interests
- **Technical approach**: Ask about implementation strategies
- **Codebase understanding**: Request clarification on existing components
- **Deployment**: Get help with production setup

> 💡 **Remember**: These features are designed to be educational. The goal is to deeply understand multi-agent systems, not just to add functionality.
>
> All the best!