**Objective**

Create a Python-based agentic chatbot leveraging the **LangGraph framework as the core orchestrator** for defining and managing the agent's stateful workflow. This system will utilize LangChain components (tools, runnables, standard memory) within the LangGraph nodes, LightRAG for efficient retrieval augmentation (including graph-based search), LiteLLM for flexible model management, and a comprehensive multi-layered memory system. The master agent's logic will be explicitly built upon this LangGraph base to interpret complex queries, orchestrate tools, and synthesize information.

**Technology Stack**

* **Primary:** Python, LightRAG
* **Supporting:** LangChain, LangGraph, LlamaIndex, LiteLLM, MongoDB (for user/conversation data and memory components), potentially other libraries as needed for specific tools or functionalities.

**Mermaid Script Requirements (for Architecture Diagrams)**

Version 2, using different colors and subgraphs. Font should be slightly larger and readable. Flow lines can be curved. MongoDB should be represented iconically (e.g., cylindrical shape). Do not display individual collections within the MongoDB representation in high-level diagrams. Use a hand-drawn look style where available and appropriate in Mermaid.

**Core Requirements**

1. **Master Agent Orchestration:**
   * The master agent serves as the central intelligence, analyzing incoming queries to determine the full scope of information or tasks required.
   * For queries involving multiple distinct requests, the agent will seek clarification from the user to confirm its understanding before proceeding.
   * After results are obtained from any tool or process, the agent will analyze, synthesize, and integrate this information into the ongoing conversation context.
   * The agent can decide to fetch information for different parts of a complex query sequentially or in parallel, combining all necessary results before generating a final, coherent response using a dedicated 'Final Response Refiner' tool.
2. **Query Routing System (Meta-Tool):**
   * Implement a sophisticated meta-tool that acts as the initial query gatekeeper. It will use an LLM to analyze incoming user queries for intent, content, and required actions.
   * Based on this analysis, it will dynamically route the query to the most appropriate specialized tool or sequence of tools within the agent's suite.
   * This system must maintain conversation context seamlessly during transitions between tools and processing steps.
3. Tool Suite:  
   Implement the following specialized tools with clear interfaces and robust error handling:  
   * **a) Stock Information Tool:**
     + Fetch real-time or historical stock ticker information (e.g., from Yahoo Finance API).
     + Process, format, and present financial data in a user-friendly manner.
   * **b) Account API Tool:**
     + Interface securely with the local endpoint: http://localhost:8080/api/account/v1.
     + Handle authentication, request formatting, and response parsing.
     + Implement comprehensive error handling for API interactions.
   * **c) Enhanced Search Tool:**
     + Utilize an LLM to rephrase and optimize search queries based on the current conversation context and user intent.
     + Perform searches and return relevant information, including source attribution.
     + Integrate results into the agent's context for synthesis.
   * **d) Document Processing Tool:**
     + Extract and process text content from various document formats (PDF, DOCX, TXT) using LightRAG's document processing capabilities. Fall back to LangChain's document loaders if necessary.
     + Prepare document content for RAG or analysis.
   * **e) Response Refinement Tool ('Final Response Refiner'):**
     + This tool receives the synthesized information and the original query context.
     + Apply tone adjustments, ensure coherence, and summarize lengthy content while preserving key details.
     + Critically assess the response for completeness and hallucination, especially when combining RAG and tool outputs. Adjust LLM parameters (e.g., temperature) for different response requirements (e.g., stringent factual responses vs. creative text).
4. Memory and Context Management:  
   Implement a multi-layered memory system to provide the chatbot with robust conversational history and learning capabilities.  
   * **Short-Term Memory:**
     + Maintain the immediate conversation history within a single user session (e.g., using ConversationBufferMemory).
     + Ensure context persistence across immediate query-response turns.
     + Implement efficient token management to prevent context overflow in active conversations.
   * **Episodic Memory:**
     + Store detailed records of specific interactions, including the user query, the agent's internal reasoning process, tool usage, tool outputs, intermediate steps, and the final generated response for each turn within a conversation.
     + This provides a rich history of *how* a particular outcome was reached. Store this data linked to the conversation in MongoDB.
   * **Long-Term Agentic Memory:**
     + Store learned insights, distilled facts, refined user preferences (beyond initial profile settings), successful tool-use patterns for common queries, and general knowledge gained from interactions that is deemed valuable for future conversations.
     + This memory should enable the agent to improve its performance and personalization over time.
     + **Background Update System:** Implement an asynchronous or periodic process that analyzes recent conversation history (drawing from Episodic Memory) to identify patterns, extract potentially valuable insights, and update the Long-Term Agentic Memory store without impacting the real-time conversation flow. This system should handle de-duplication and relevance assessment for long-term storage.
   * **Database Integration:**
     + Use MongoDB to store user information (userId, firstName, lastName, loginId, preferences).
     + Store conversation history in a conversations collection. Each document will contain conversationId, userId, conversation\_datetime, and a detailed record of the conversation turns (including query, response, and potentially a link or embedded structure for the Episodic Memory details of that turn).
5. **RAG Implementation:**
   * Utilize LightRAG as the primary framework for Retrieval Augmented Generation.
   * LightRAG should dynamically decide the appropriate RAG mode (e.g., mix, global, local, naive) based on the query complexity and available information sources.
   * Implement graph-based entity search capabilities using LightRAG when the query involves specific entities and relationships.
   * Integrate with a suitable vector store for efficient retrieval of relevant document chunks.
   * Apply reranking techniques to improve the relevance of retrieved results before synthesis.
   * Generate responses using LLMs, carefully combining retrieved information with conversation context and agent reasoning.
6. **Agent Architecture:** 
   * Implement the agent's core reasoning and action flow as a stateful graph using **LangGraph**. LangGraph will serve as the primary orchestrator and base framework, defining the agent's states, nodes (representing actions like tool calls, RAG steps, decision points), and edges (dictating transitions based on state and outcomes). The ReAct pattern will be implemented *within* this LangGraph structure.
   * LangChain components, including Tools, LCEL runnables, and interactions with the memory system, will be integrated and utilized **as the operational logic within the nodes** of the LangGraph.
   * The ReAct reasoning happens **in a node**, e.g., decide\_next\_node similar to the ReAct (Reasoning and Action) pattern to guide the agent's decision-making process and tool usage.
   * Consider using custom LangGraph implementation for handling multi-step reasoning and complex query execution flows.

**Implementation Guidelines**

* LangGraph provides the fundamental structure and state management for the agent's dynamic workflow, orchestrating the execution of steps defined using LangChain components and other libraries.
* Prioritize utilizing LightRAG's native functionalities for RAG and document processing before incorporating other libraries for similar tasks.
* Use LiteLLM to abstract access to various LLMs and providers, ensuring flexibility and ease of model switching.
* Leverage LangChain's robust agent framework, tool definitions, and memory interfaces where appropriate.
* Incorporate LlamaIndex only for specific, unique functionalities not readily available or more efficiently implemented in LightRAG or LangChain.
* Ensure seamless integration between LightRAG, LangChain/LangGraph, and the custom memory components.

**Code Requirements**

* Include complete and correct import statements for all utilized libraries and modules.
* Use the latest stable versions of all packages, ensuring compatibility between them.
* Implement comprehensive error handling throughout the application, providing graceful degradation and informative logging.
* Include clear, concise comments explaining component interactions, complex logic, and design choices.
* Structure the codebase in a modular, maintainable, and scalable manner.

**Evaluation Criteria**

The final implementation will be evaluated on:

* Correct routing of queries to appropriate tools and sequences.
* Effective and demonstrable use of RAG (via LightRAG) to enhance response quality.
* Coherent and contextually aware multi-turn conversations, showcasing effective use of Short-Term, Episodic, and Long-Term memory.
* Proper functioning of the background Long-Term memory update system.
* Robust error handling and graceful degradation under unexpected conditions.
* Code quality, modularity, documentation, and adherence to best practices.
* Successful implementation of the defined tool suite.
* Proper data storage and retrieval from MongoDB for user and conversation history.

**Project Phases (for phased development)**

We will proceed in distinct phases, creating architecture diagrams for each and seeking confirmation before moving to the next.

1. **Phase 1: Basic Setup & Core Backend:**
   * Set up the basic project structure, including UI (placeholder/minimal) and a backend API endpoint.
   * The backend accepts a query and returns a simple response generated directly by an LLM (via LiteLLM).
   * Implement basic MongoDB connection and user/conversation data models (without complex history storage yet).
   * *Architecture Diagram:* High-level flow from UI to Backend to LLM, including Database connection.
2. **Phase 2: Introduce LangChain Agent & First Tool:**
   * Integrate LangChain and set up a basic agent structure.
   * Implement the Stock Information Tool (fetching from Yahoo Finance API).
   * Configure the agent to decide *when* to use the Stock tool based on query intent.
   * Implement basic Short-Term Memory using LangChain's capabilities.
   * *Architecture Diagram:* Detail the agent's interaction with the LLM, the Stock Tool, and Short-Term Memory.
3. **Phase 3: Add Second Tool & Agent Tool Selection:**
   * Implement the Weather Data Tool (fetching data based on zip code/city/state).
   * Enhance the agent's logic to choose between the Stock and Weather tools based on query analysis.
   * *Architecture Diagram:* Update the diagram to show the agent selecting between two tools.
4. **Phase 4: Introduce Vector DB Search Tool:**
   * Set up a simple vector database (e.g., using SQLite and an embedding model).
   * Implement the RAG Vector DB Search Tool with an available API endpoint.
   * Configure the agent to decide when to use the Vector DB Search tool.
   * *Architecture Diagram:* Add the Vector DB and the new tool to the agent diagram.
5. **Phase 5: Basic RAG Integration & Context Refinement:**
   * If the RAG Vector DB Search tool is used, integrate the retrieved results into the agent's context.
   * Implement a basic process where an LLM refines the context by incorporating RAG results before the final response generation.
   * *Architecture Diagram:* Illustrate the flow involving Vector DB retrieval, result integration, and LLM context refinement.
6. **Phase 6: Advanced Agent Reasoning & Tool Chaining:**
   * Implement the core agent logic to analyze multi-part queries, determine required tools and their sequence.
   * Develop the capability for the agent to execute a chain or graph of tool calls to fulfill complex requests.
   * Focus on the agent's ability to gather results from multiple sources and synthesize them.
   * *Architecture Diagram:* Depict the agent's internal reasoning, planning, and multi-step tool execution flow (potentially introducing LangGraph concepts here).
7. **Phase 7: Implement 'Final Response Refiner' Tool:**
   * Implement the dedicated 'Final Response Refiner' tool.
   * Ensure it checks response completeness, controls hallucination, and adjusts LLM parameters based on content type (e.g., strict factual data vs. conversational text).
   * Integrate this tool as the final step before presenting the response to the user.
   * *Architecture Diagram:* Show the final step where synthesized information goes to the Refiner tool before output.
8. **Phase 8: UI Development (React) & History Features:**
   * Develop the React-based UI with the specified functionalities:
     + Option to choose LLM provider and specific model (leveraging LiteLLM).
     + Display and store chat conversation history (using backend APIs).
     + Ability to select, rename, and delete previous conversation history.
     + Ability for the user to set preferences (stored in MongoDB).
   * Develop necessary backend APIs to support these UI features (interacting with MongoDB).
   * *Architecture Diagram:* Focus on the UI layer, its components, and its interaction with the backend APIs and database.
9. **Phase 9: Deep Memory Integration & Background Updates:**
   * Fully integrate Episodic Memory storage (detailed interaction steps per turn) into the MongoDB conversation structure.
   * Implement the Long-Term Agentic Memory store and its structure in MongoDB or a separate suitable store.
   * Develop the background process for analyzing conversation history (Episodic Memory) and updating the Long-Term Agentic Memory with learned insights and patterns.
   * Refine the agent's logic to utilize Episodic and Long-Term memory to inform its decisions and responses.
   * *Architecture Diagram:* Detail the interaction between different memory types (Short-Term, Episodic, Long-Term), the agent, the database, and the new background update process.
10. **Phase 10: Observability & LightRAG Advanced Features:**
    * Set up OpenTelemetry (Otel) with a backend like Phoenix or Jaeger to trace requests (@Trace annotation equivalent if applicable) and store interaction flows and performance data for debugging and monitoring.
    * Integrate advanced LightRAG features: dynamic mode selection (mix, global, local, naive), graph-based entity search.
    * Implement LightRAG's Document Processor with a UI component for uploading files (PDF, TXT) and adding metadata.
    * Implement LightRAG's web scraping capability for document processing.
    * *Architecture Diagram:* Add the observability component (Otel/Phoenix), detail LightRAG's advanced RAG pipeline, and show the document processing/web scraping flow.

**Post-Phase Architecture Diagrams**

Once all development phases are diagrammed, we will create the following overarching architecture diagrams for the complete system:

* High-Level System Architecture Diagram
* Data Flow Diagram (DFD)
* Component Level Architecture Diagram
* RAG/LLM-specific Architecture Diagram (detailing LightRAG, Vector Store, LLMs, etc.)
* Security Architecture Considerations Diagram
* Infrastructure Diagram (if deployed on GCP, detailing Services, Gemini API, Vertex AI)
* Infrastructure Diagram (if deployed on AZURE, detailing Compute Services, AI Service, Database)
* Infrastructure Diagram (if deployed on Amazon AWS, detailing Compute Services, AI Service, Database)

This refined prompt incorporates your requirements for different memory types and the background update system, integrating them into the existing detailed plan. It maintains the structure necessary for proceeding through the defined phases.

Let me know if this refined prompt meets your requirements for the project description. Once you confirm, we can proceed with generating the diagrams for Phase 1.

Confidence Level: 5/5