Agentic RAG System – System Design Document

1. System Architecture

The Agentic RAG System is designed using a modular, layered architecture to ensure scalability, maintainability, and clear separation of concerns. The system consists of a user-facing interface, a backend API layer, an agentic reasoning engine, and a retrieval layer backed by a vector database.

The Streamlit-based UI allows users to submit natural language queries. These queries are forwarded to a FastAPI backend, which acts as a thin orchestration layer. The backend invokes the Agentic RAG engine implemented using LangGraph and Groq LLM. Based on agent decisions, relevant tools retrieve context from the vector database, and a grounded response is generated.

2. Agentic Workflow Design

The system follows an agentic workflow rather than a static retrieval pipeline. Upon receiving a query, an AI agent first reasons about the intent of the query. The agent dynamically decides whether retrieval is required and selects the most relevant tool at runtime.

This decision-making process is autonomous and not governed by hardcoded rules. The agent controls the flow of execution, enabling conditional retrieval and flexible reasoning paths. This design demonstrates true agentic behavior, where the LLM governs when and how external knowledge is accessed.

3. Context Construction Strategy

Context construction is performed through Retrieval-Augmented Generation. Source documents such as PDFs, Excel files, and text files are preprocessed, chunked, and converted into vector embeddings. These embeddings are stored in a vector database to enable semantic search.

When the agent selects a tool, the tool retrieves the most semantically relevant document chunks. Only these chunks are passed as context to the LLM. This approach ensures concise, relevant, and grounded context, significantly reducing hallucinations in the generated responses.

4. Technology Choices and Rationale

Groq LLM was chosen for fast and cost-effective inference. LangGraph was used to model agentic workflows and manage multi-step reasoning. FastAPI was selected for its performance and ease of integration, while Streamlit was used to rapidly build an interactive user interface.

Docker was used to containerize the application, enabling consistent deployments across environments. AWS EC2 was chosen as the deployment platform due to its flexibility and simplicity. Vector storage was implemented using a vector database to support efficient semantic retrieval.

5. Key Design Decisions

A key design decision was to separate the UI, backend API, and agentic reasoning layers. This enables independent scaling and easier maintenance. Another important decision was to avoid hardcoded routing logic and instead allow the agent to dynamically select tools.

The system also avoids embedding secrets into containers, relying instead on environment variables. Additionally, the application was designed to be cloud-ready without mandating complex infrastructure such as Kubernetes or managed services.

6. Limitations

The current implementation does not include MCP (Model Context Protocol) servers. While MCP was considered, direct tool integration was chosen to reduce complexity. Chat history persistence is also not implemented in this version.

Future enhancements could include MCP integration, advanced reranking strategies, persistent memory, and enhanced access control mechanisms.