## **Development Plan: Agentic RAG Chatbot**

**Part 1: Foundational Elements**

Before diving into phased development, ensure these are in place or planned for early setup:

1. **Core Programming Languages & Frameworks:**
   * **Backend & Agent Logic (Python):**
     + FastAPI: For building robust and efficient APIs.
     + LangGraph: As the core orchestrator for agent state and workflow.
     + LangChain: For LLM integrations, tool definitions, basic memory components, LCEL.
     + LightRAG: For the RAG pipeline, document processing, and advanced RAG features.
     + LiteLLM: For abstracting calls to OpenAI and Anthropic (Claude) LLMs, and potentially for embedding models if not using dedicated ones.
   * **Frontend (JavaScript/TypeScript):**
     + React: For building the user interface.
     + State Management: (e.g., Redux, Zustand, Context API)
     + HTTP Client: (e.g., Axios, Fetch API)
   * **Databases (Open Source):**
     + Document/Metadata/LTM Store: MongoDB (Community Edition) or PostgreSQL (with JSONB).
     + Vector Store (RAG KB, Episodic Memory): PostgreSQL (with pgvector extension), Milvus, Weaviate, or Qdrant.
     + Cache (STM): Redis (Open Source).
2. **Development Environment & Tools:**
   * **Version Control:** Git (GitHub, GitLab, or self-hosted Gitea/GitLab). Establish branching strategy (e.g., Gitflow, Trunk-based).
   * **IDE:** VS Code (with Python, Docker, Pylance, Prettier extensions) or PyCharm Professional.
   * **Containerization:** Docker (for all services) and Docker Compose (for local development environment setup).
   * **Package Management:**
     + Python: Poetry (recommended for dependency management and packaging) or pip with requirements.txt.
     + JavaScript/TypeScript: npm or yarn.
   * **Virtual Environments (Python):** Poetry automatically handles this; otherwise, venv.
   * **Linters & Formatters:**
     + Python: Black (formatter), Flake8 or Ruff (linter).
     + JavaScript/TypeScript: ESLint (linter), Prettier (formatter).
     + Consider pre-commit hooks to automate these.
   * **Testing Frameworks:**
     + Python: Pytest (for unit, integration, and API tests). pytest-asyncio for FastAPI.
     + JavaScript/TypeScript: Jest, React Testing Library.
   * **API Documentation:** OpenAPI (Swagger) automatically generated by FastAPI.
   * **Task Management:** A project management tool (Jira, Trello, Asana, or even GitHub Issues/Projects).
3. **Cross-Cutting Best Practices:**
   * **Modular Design:** Break down the system into loosely coupled, highly cohesive modules/services (as per our component diagrams).
   * **Iterative Development:** Follow the 10 phases as iterative sprints, seeking feedback and refining.
   * **Comprehensive Testing:**
     + Unit tests for individual functions/classes.
     + Integration tests for interactions between components (e.g., API to Agent Core, Agent Core to RAG).
     + End-to-End (E2E) tests for user flows (e.g., UI to final response).
   * **CI/CD (Continuous Integration/Continuous Deployment):**
     + Set up early (e.g., GitHub Actions, GitLab CI, Jenkins).
     + Automate linting, testing, building Docker images, and deploying to dev/staging environments.
   * **Documentation:**
     + Code Comments: Docstrings for Python functions/classes, JSDoc for JS/TS.
     + READMEs: For each service/repository and the overall project.
     + Design Documents: Maintain and update the diagrams and prompt we've created.
     + API Documentation: Auto-generated via FastAPI.
   * **Secrets Management:** Use HashiCorp Vault (Open Source version) for storing API keys (OpenAI, Claude, external tools), database credentials, etc. Inject these into containers as environment variables, not hardcoded.
   * **Configuration Management:** Separate configuration from code (e.g., using environment variables, config files).
   * **Security by Design:**
     + Follow OWASP Top 10 guidelines.
     + Implement security controls identified in the Security Architecture diagram.
     + Regularly update dependencies.

**Part 2: Step-by-Step Development Approach (Iterating Through the 10 Phases)**

This section translates our architectural phases into a development roadmap. For each phase, focus on building the specified functionality and then integrating it into the open-source infrastructure.

**Sprint 0: Foundation & Setup**

* **Goal:** Establish the core development environment, infrastructure basics, and CI/CD pipeline.
* **Tasks:**
  1. Set up Git repository.
  2. Set up local development environments (Docker Compose for databases, Redis).
  3. Basic FastAPI backend project structure.
  4. Basic React frontend project structure.
  5. Initial CI/CD pipeline (linting, basic tests).
  6. Set up LiteLLM locally to connect to OpenAI/Claude with placeholder API keys (managed via a local .env for now, to be moved to Vault later).
  7. Initial setup of Kubernetes (e.g., Minikube/K3s for local/dev) if taking a K8s-first approach for services.
  8. Deploy initial open-source databases (MongoDB, PostgreSQL+pgvector, Redis) on the dev K8s cluster or locally via Docker Compose.
  9. Set up HashiCorp Vault for secrets.

**Phase 1 (Development Sprint 1): Basic Setup & Core Backend**

* **Goal:** Minimal UI to Backend API flow with direct LLM response. Basic MongoDB setup.
* **Dev Tasks:**
  1. **Backend:**
     + Implement a FastAPI endpoint (/api/v1/chat/simple) that takes a query.
     + Integrate LiteLLM to route this query to a chosen LLM (OpenAI or Claude).
     + Return the raw LLM response.
     + Implement basic MongoDB connection:
       - Define Pydantic models for User and basic ConversationLog.
       - Basic functions to save user info (placeholder) and the simple query/response to MongoDB.
  2. **Frontend:**
     + Create a very basic React UI with an input field and a display area for the response.
     + Call the /api/v1/chat/simple endpoint.
  3. **Infrastructure:**
     + Dockerize the FastAPI backend and React UI.
     + Deploy to local K8s/Docker Compose.
     + Ensure connectivity to MongoDB and LiteLLM (pointing to external LLMs).
* **Deliverable:** A working UI that can send a query to the backend, get a direct LLM response, and log the interaction.

**Phase 2 (Development Sprint 2): Introduce LangGraph & First Tool**

* **Goal:** Integrate LangGraph with a simple graph for one tool (Stock). Basic STM.
* **Dev Tasks:**
  1. **Backend (Agent Core Service):**
     + Design and implement the initial LangGraph state.
     + Create a LangGraph with nodes:
       - receive\_query: Takes input.
       - route\_to\_tool\_or\_direct: Simple LLM call via LiteLLM to decide if StockTool is needed.
       - call\_stock\_tool: A LangChain tool for fetching stock info (initially can be a mock, then integrated with Yahoo Finance API).
       - direct\_llm\_response: If no tool, use LLM for response.
       - format\_response.
     + Integrate basic LangChain ConversationBufferMemory into the LangGraph state for STM.
     + Expose this LangGraph via a new FastAPI endpoint (e.g., /api/v1/chat/agent).
  2. **Tooling Layer:**
     + Implement the StockInformationTool (LangChain BaseTool).
  3. **Frontend:** Update UI to call the new agent endpoint.
  4. **Infrastructure:** Update deployments. LiteLLM is used by the router node.
* **Deliverable:** Agent can answer direct questions or use the Stock tool. STM maintains context for a short conversation.

**Phase 3 (Development Sprint 3): Add Second Tool & LangGraph Routing**

* **Goal:** Enhance LangGraph routing to choose between Stock and Weather tools.
* **Dev Tasks:**
  1. **Backend (Agent Core Service):**
     + Modify route\_to\_tool\_or\_direct node in LangGraph to use an LLM (via LiteLLM) to classify intent for Stock, Weather, or direct LLM.
     + Add call\_weather\_tool node.
  2. **Tooling Layer:**
     + Implement the WeatherDataTool (LangChain BaseTool), interfacing with a weather API.
  3. **Infrastructure:** Ensure secrets for the weather API are in Vault.
* **Deliverable:** Agent can now use Stock or Weather tools based on query intent.

**Phase 4 (Development Sprint 4): Introduce Vector DB Search Tool Node**

* **Goal:** Set up Vector DB. Agent can route to a RAG search tool.
* **Dev Tasks:**
  1. **Data Tier:**
     + Set up PostgreSQL with pgvector (or Milvus/Weaviate/Qdrant).
     + Manually ingest a few sample documents (embeddings generated via a script using, e.g., OpenAI embeddings through LiteLLM or a SentenceTransformer model).
  2. **Backend (Agent Core Service):**
     + Update route\_to\_tool\_or\_direct to include RAG search as an option.
     + Add call\_rag\_search\_tool node in LangGraph.
  3. **Tooling Layer:**
     + Implement VectorDBSearchTool that queries the Vector DB.
* **Deliverable:** Agent can perform a basic RAG search if query intent matches.

**Phase 5 (Development Sprint 5): Basic RAG Integration & Context Refinement Node**

* **Goal:** Integrate RAG results into context using an LLM.
* **Dev Tasks:**
  1. **Backend (Agent Core Service):**
     + Add a refine\_context\_with\_rag node in LangGraph after call\_rag\_search\_tool.
     + This node takes RAG results and the current query/STM, uses an LLM (via LiteLLM) to synthesize, and updates the LangGraph state.
* **Deliverable:** If RAG is used, retrieved content refines the context before final response generation.

**Phase 6 (Development Sprint 6): Advanced LangGraph Reasoning & Tool Flow**

* **Goal:** Full agent LangGraph for multi-part queries, tool sequencing, state management for accumulated results.
* **Dev Tasks:**
  1. **Backend (Agent Core Service):**
     + Overhaul the LangGraph design based on the Phase 6 diagram:
       - Parse & Deconstruct Query Node: LLM-based analysis for sub-tasks.
       - Task Planner Node: Sequences tool/RAG calls.
       - Nodes for parallel/sequential execution paths (conceptually, actual parallelism later if needed).
       - Nodes for updating/accumulating results in the LangGraph state.
       - Synthesize All Collected Results Node.
* **Deliverable:** Agent can handle more complex queries requiring multiple steps or tools.

**Phase 7 (Development Sprint 7): Implement 'Final Response Refiner' Node**

* **Goal:** Dedicated node for final response polishing.
* **Dev Tasks:**
  1. **Backend (Agent Core Service):**
     + Implement the FinalResponseRefinerTool (as a LangChain tool or a dedicated LangGraph node).
     + This node uses an LLM (via LiteLLM) to check completeness, tone, hallucination, and apply company regulation checks (initial version).
     + Integrate this as a terminal or near-terminal node in the main LangGraph.
* **Deliverable:** Responses are more polished and undergo a final check.

**Phase 8 (Development Sprint 8): UI Development (React) & History APIs**

* **Goal:** Feature-rich UI with LLM selection, full history management, and preferences.
* **Dev Tasks:**
  1. **Frontend:**
     + Develop React components for LLM selection, history panel (view, rename, delete), and user preferences modal.
  2. **Backend (API Layer - FastAPI):**
     + Implement /api/history CRUD endpoints.
     + Implement /api/models endpoint (interfacing with LiteLLM to list configured models).
     + Implement /api/preferences CRUD endpoints.
  3. **Database (MongoDB):**
     + Finalize schemas for User (with preferences) and Conversations collections.
* **Deliverable:** Fully functional UI as per Phase 8 specs.

**Phase 9 (Development Sprint 9): Deep Memory Integration & Background Updater**

* **Goal:** Implement Episodic and Long-Term Memory storage and the background LTM updater.
* **Dev Tasks:**
  1. **Backend (Agent Core Service & Memory Management Layer):**
     + Modify LangGraph nodes to log detailed interaction traces (inputs, reasoning steps, tool calls/outputs) to the Vector DB (Episodic Memory).
     + Link these traces to conversation logs in MongoDB.
     + Implement LTM read access in relevant LangGraph nodes (e.g., query planner).
  2. **Backend (LTM Updater Service - Separate LangGraph):**
     + Develop the scheduled LangGraph subgraph:
       - Nodes to read Episodic Memory (VectorDB, MongoDB logs).
       - Nodes for LLM-based analysis to extract insights.
       - Nodes to consolidate and write insights to LTM store in MongoDB.
  3. **Infrastructure:**
     + Set up scheduler (Cron in K8s, or Open Source Airflow if complex) to trigger the LTM updater.
* **Deliverable:** Agent stores detailed episodic memory. LTM is populated and updated by the background process. Agent can start using LTM.

**Phase 10 (Development Sprint 10): Observability & LightRAG Advanced Features**

* **Goal:** Integrate observability and advanced LightRAG features. Full document ingestion pipeline.
* **Dev Tasks:**
  1. **Observability (Across all services):**
     + Integrate OpenTelemetry SDK.
     + Configure exporters for Jaeger/Zipkin (traces), Prometheus (metrics), Loki/ELK (logs).
     + Instrument key parts of the API, Agent Core, Tools, RAG pipeline.
  2. **Backend (RAG Coordinator & LightRAG Integration):**
     + Implement dynamic RAG mode selection node.
     + Implement graph-based entity search node using LightRAG.
  3. **Data Ingestion Layer (LightRAG Document Processor Service):**
     + Develop the full document processing pipeline (file handling, web scraping using LightRAG modules).
     + Integrate embedding generation.
     + Implement storage to Vector DB and MongoDB (metadata).
  4. **Frontend:**
     + Develop UI component for document upload & metadata input, calling the ingestion API.
  5. **Infrastructure:**
     + Deploy Prometheus, Grafana, Jaeger/Loki.
     + Ensure ingestion service can scale.
* **Deliverable:** System has observability. Advanced RAG features are active. Full document ingestion pipeline is functional.

**Part 4: Project Plan High-Level Outline**

This is a sequence of activities rather than a Gantt chart with timelines.

1. **Sprint 0: Foundations & Setup**
   * Project Repo & Version Control Strategy
   * Local Development Environment (Docker Compose: PG+pgvector, MongoDB, Redis, LiteLLM)
   * Basic CI Pipeline (Lint, Format, placeholder tests)
   * Secrets Management Setup (Vault - initial config)
   * Kubernetes Dev Cluster (Minikube/K3s) - Initial exploration
2. **Application Development (Iterative - Sprints 1-10, mapping to Phases 1-10):**
   * For each phase:
     + Detailed design of components/modules for that phase.
     + Backend development (FastAPI, LangGraph nodes, tools).
     + Frontend development (React components, API integration).
     + Unit and Integration Tests.
     + Dockerization of new/updated services.
     + Deployment to dev K8s environment.
     + Update LiteLLM configuration as new models/needs arise.
     + Update database schemas and interactions.
     + Update CI/CD pipeline.
     + Documentation.
3. **Infrastructure Maturation (Parallel with Application Sprints, focus on open-source stack):**
   * **Early Sprints (1-4):** Focus on robust Docker Compose for local dev. Begin deploying core DBs (PG, Mongo, Redis) to dev K8s. Basic Nginx for UI.
   * **Mid Sprints (5-8):** Solidify K8s deployments. Set up API Gateway (Kong/Nginx). Start implementing Observability stack (Prometheus, Grafana, Jaeger/Loki). Integrate Vault more deeply.
   * **Late Sprints (9-10):** Mature Observability. Implement scheduling for LTM updater (K8s CronJob or Airflow). Finalize production-like K8s configurations (networking, storage, scaling).
4. **Testing & QA (Continuous & Dedicated Phases):**
   * Ongoing: Unit, integration tests with each feature/sprint.
   * Dedicated QA Sprints: After major milestones (e.g., after Phase 6, after Phase 8, after Phase 10). Focus on E2E testing, performance testing, security testing.
5. **Deployment & Go-Live Prep:**
   * Staging environment setup (mirroring production open-source stack).
   * Full E2E testing on staging.
   * User Acceptance Testing (UAT).
   * Finalize production deployment scripts/automation for Kubernetes.
   * Data migration/initial seeding if necessary.
   * Monitoring and alerting setup (Grafana).
6. **Post-Launch:**
   * Monitoring, maintenance, bug fixing.
   * Iterative improvements based on user feedback and LTM insights.

**Key Milestones (can align with groups of phases):**

* **M1 (End of Phase 5):** Core agent with basic RAG and multiple tools functional.
* **M2 (End of Phase 8):** Full UI functionality with robust agent core.
* **M3 (End of Phase 10):** Feature-complete system with memory, observability, and advanced RAG. Ready for intensive E2E testing and UAT.

This comprehensive plan should provide a solid roadmap. Remember that agility is key; be prepared to adjust the plan as you learn more during development. Regular demonstrations and feedback loops with stakeholders will be crucial.