Design and Architecture of an Adaptive Multi-Agent Chatbot System

Executive Summary

This report presents the design and implementation of an **Adaptive Multi-Agent Chatbot System** built in Python. The system leverages multiple specialized AI agents, each responsible for a distinct domain of user queries. A local Large Language Model (LLM) is deployed via **Ollama**, enabling on-device inference for privacy and low-latency interactions. The architecture is modular, with dedicated agents for general knowledge Q&A, Concordia University's Computer Science admissions inquiries, and AI-specific questions. A central orchestrator component routes user queries to the appropriate agent, enabling context-appropriate responses. The chatbot maintains context over multi-turn dialogues using **LangChain** memory mechanisms and a **FAISS** vector store for efficient retrieval of relevant conversation history and domain knowledge. External knowledge sources (e.g. Wikipedia) are integrated to enhance responses with up-to-date information. The system is exposed through a **FastAPI** web service for easy deployment and interaction.

In summary, the project demonstrates how combining local LLM inference, a multi-agent architecture, and retrieval-augmented techniques can create a robust conversational agent. We detail the system's overall architecture, the rationale behind technology choices (Ollama, FastAPI, LangChain, FAISS, etc.), design trade-offs, implementation challenges encountered (from coordinating multiple agents to managing dependencies), and future improvements. By building the chatbot with modern frameworks and a clear modular structure, we achieved a flexible platform that can be extended with new agents or scaled for higher performance. Key strengths of the system include its privacy-preserving local model, specialized domain expertise via multiple agents, and context-aware dialogue handling. Areas for improvement include optimizing inference speed

(possibly with GPUs), better scalability, and continued refinement of agent coordination and knowledge integration.

Architectural Overview

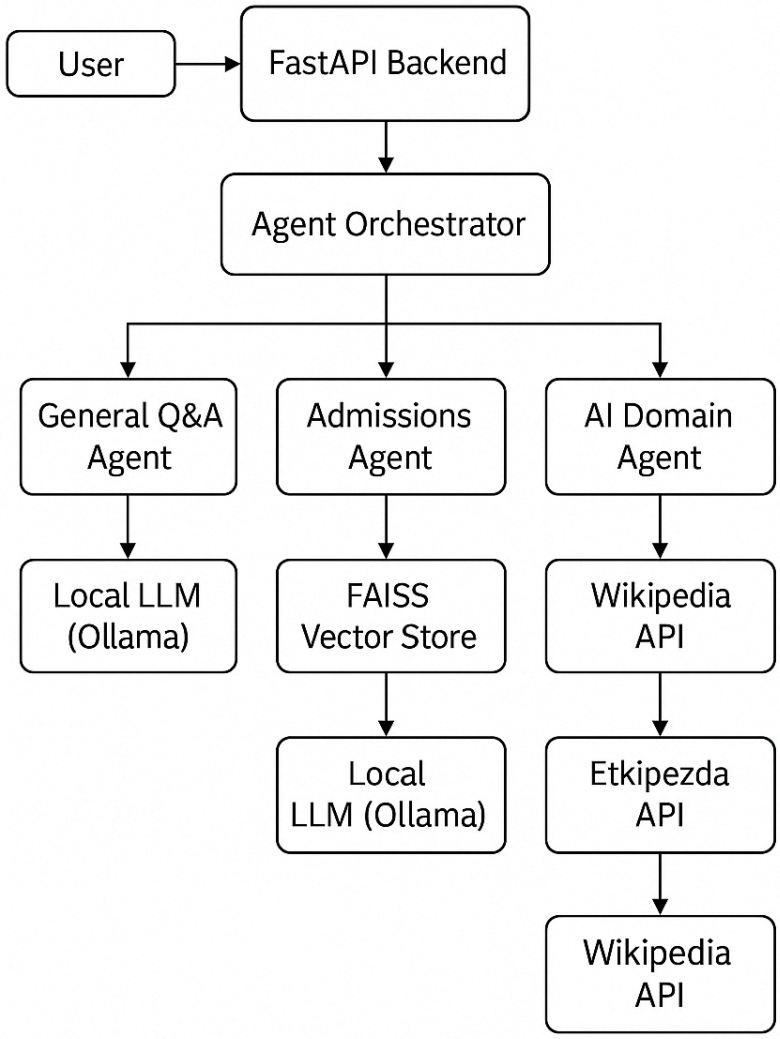

Figure 1: High-level architecture of a multi-agent chatbot system

Figure 1: High-level architecture of the multi-agent chatbot system. The user's query is processed by a FastAPI backend and routed by an Agent Orchestrator to one of several specialized agents (General Q&A, Admissions, or AI domain). Each agent may utilize the local LLM (via Ollama) for natural language understanding and generation, query a FAISS vector store for relevant context, and fetch external information from the Wikipedia API if needed. The selected agent's response is returned to the user through the FastAPI service.

The architecture consists of three main layers: (1) the **API interface layer**, (2) the **multi-agent core**, and (3) the **supporting knowledge and memory systems**. At the top layer, a FastAPI web service handles incoming HTTP requests from users and returns chatbot responses. FastAPI was chosen due to its high performance and ease of building production-ready RESTful APIs in Python (FastAPI). Each user query (e.g., a chat message) is received by the FastAPI app and forwarded to the multi-agent core logic for processing.

The **multi-agent core** contains an **Agent Orchestrator** that manages multiple specialized chatbot agents. Upon receiving a query, the orchestrator determines which agent is best suited to handle it. The system supports at least three distinct agents:

- **General Q&A Agent:** Handles open-domain questions or casual conversation.
- **Admissions Agent:** Expert in Concordia University's Computer Science admissions details (e.g., program requirements, application process).
- **AI Expert Agent:** Focused on artificial intelligence topics, capable of answering technical AI questions or discussion.

The Agent Orchestrator can use a routing algorithm or classifier to decide which agent to invoke. For example, admissions-related keywords might trigger the Admissions Agent. In other cases, the orchestrator might employ a router chain (a LangChain component) that uses an LLM to classify the query's domain and route accordingly. This design allows the chatbot to **adapt** to different query types dynamically by activating the relevant agent, hence the "adaptive multi-agent" moniker.

Each specialized agent operates with its own **prompt context, tools, and memory**. Agents interact with the underlying LLM (provided by Ollama) to generate responses. They also can incorporate external knowledge: for instance, the General or AI agent may call the Wikipedia API (through a utility or LangChain tool) to retrieve up-to-date facts on a topic, supplementing the LLM's built-in knowledge. The Admissions agent might use an internal knowledge base (documents or FAQs about the university) stored in the vector database for reference. All agents share access to a **common memory store** backed by FAISS, which retains embeddings of conversation history or domain-specific data for retrieval. This memory system ensures that even if the conversation switches

between agents, important context from previous interactions can be retrieved and passed into the LLM prompt, enabling continuity across turns.

The bottom layer of the architecture includes the **local LLM inference engine** and **knowledge stores**. The LLM is hosted via Ollama, which runs a model (such as a Llama 3 variant) locally. Ollama provides a server endpoint on localhost for generating completions from the model. By using Ollama, we avoid external API calls and keep user data on-premise, addressing privacy, latency, and cost concerns associated with cloud models (Ollama Cheatsheet: Running LLMs Locally with Ollama - DEV Community). The vector store (FAISS) is employed for both short-term conversational memory and long-term knowledge retrieval. When an agent needs to recall previous context or fetch relevant background information, it encodes the query or conversation turn into an embedding and performs a similarity search in FAISS. The top matching vectors (which could be past user questions, prior answers, or reference documents) are then used to augment the prompt given to the LLM. This retrieval-augmented approach grounds the LLM's responses in relevant data, reducing hallucinations and improving factual accuracy. Meanwhile, the **Wikipedia integration** is done through the MediaWiki API, which is a web service allowing programmatic access to wiki content and search features (API:Main page - MediaWiki). The system can send an HTTP request to Wikipedia (or use a library like `wikipedia` in Python) to get a summary of a topic, which the agent then feeds into the LLM's context when formulating an answer.

In summary, the architecture cleanly separates concerns: FastAPI for the interface, an orchestrator and multiple agents for the core logic, and dedicated subsystems for memory (FAISS vector store), LLM inference (Ollama), and external knowledge (Wikipedia API). This modular design makes the system extensible — new agents specializing in other domains can be added without affecting the rest of the system, and components can be upgraded (for example, swapping the LLM or adding a GPU server) with minimal changes to the overall pipeline.

Detailed Design Decisions

Technology Stack and Rationale

Ollama (Local LLM Inference): We chose Ollama as the platform for running LLMs locally. **Ollama** is a user-friendly interface and engine for executing large language models on local hardware. It allows the chatbot to run offline and ensures that user data never leaves the host machine. This addresses privacy concerns and eliminates network latency in model queries. In our case, we pulled a suitable pre-trained model (such as Llama3) via Ollama and used its API to generate chat responses. The trade-off with local inference is that it can be slower or less powerful than cloud offerings (depending on hardware and model size), but for an academic prototype, the privacy and cost benefits outweighed peak performance. Additionally, using Ollama gave us flexibility to experiment with different open-source models by simply downloading them, and it supports GPU acceleration out-of-the-box to speed up inference when available however we opted for running it on CPU due to some errors and time constraints. (Running LLMs locally with Ollama. Introduction | by Omar Alva | Medium) (Running LLMs locally with Ollama. Introduction | by Omar Alva | Medium).

FastAPI (Web Framework): The system uses FastAPI to serve the chatbot as a web service. **FastAPI** was selected due to its high performance and modern features. It is a **"modern, fast (high-performance) web framework for building APIs with Python"** (FastAPI), which makes it ideal for a chatbot backend that needs to handle multiple concurrent requests with low latency. By using FastAPI, we also gain automatic interactive API documentation (via Swagger UI) and data validation through Pydantic, making our API more robust during development and testing. An alternative could have been Flask or Django, but Flask is not async-native and Django is heavier than needed. FastAPI provided the right balance of speed and ease of use, and it integrates well with our other components.

LangChain (Agent and Memory Framework): We utilized **LangChain** as the backbone for agent orchestration and memory management. LangChain is a framework that simplifies development of LLM-powered applications by providing standard interfaces for LLMs, tools, chains, and memory (LangChain - Wikipedia). It allowed us to create agents with specific prompt templates and tool usage without writing everything from scratch. For example, we leveraged LangChain's **RetrievalQA** chain for the Admissions agent to combine the LLM with vector store retrieval, and LangChain's agent tools for enabling Wikipedia searches. LangChain's **multi-agent support** (via utilities like `AgentRouter` or the newer LangChain Expression Language / LangGraph) influenced our design of how

agents communicate. The primary benefit is that LangChain provides off-the-shelf implementations for conversational memory, which we employed to handle multi-turn context. It also has an integration for Ollama models, so we could plug our local model into a LangChain LLM wrapper easily. One trade-off in using LangChain is that it can add abstraction layers that obscure what the model is doing, so we had to ensure we understood how prompts are constructed and how memory is inserted. Nonetheless, using this framework accelerated development significantly and provided a clear "cognitive architecture" for the chatbot system.

FAISS (Vector Store for Memory): For long-term memory and knowledge retrieval we chose **FAISS (Facebook AI Similarity Search)**, an open-source library optimized for fast vector similarity search (Welcome to Faiss Documentation — Faiss documentation). FAISS enables us to store text embeddings (from conversation turns or documents) and quickly retrieve the most similar ones to a new query vector. In our system, each piece of information (a user utterance, an agent response, or a chunk of external data) can be transformed into a numerical embedding vector. These vectors are indexed by FAISS, allowing cosine or L2 distance queries to find relevant past information. We used FAISS through LangChain's integration, which treats the vector store as a retriever component. **FAISS was chosen** because it runs in-memory and can scale to fairly large datasets, and it's highly efficient for our use case of moderate-sized embedding collections. It is a C++ library with Python bindings and is widely used in the industry for similarity search (Welcome to Faiss Documentation — Faiss documentation). Alternatives considered included **ChromaDB** (another simple vector DB) or cloud services like Pinecone. We opted for FAISS due to its performance and the desire to keep all components self-hosted. A potential trade-off is that FAISS operates in-memory (unless we manually persist to disk), so memory usage needs to be monitored. However, for an academic project scale, this was not an issue. The FAISS index allows the chatbot to **remember context** beyond the immediate conversation window by retrieving semantically similar past interactions. This design is crucial for context-aware dialogue: even if the conversation spans many turns, the system can remind itself of relevant details (like the user's name or preferences mentioned 10 turns ago) by querying the vector store.

Wikipedia API Integration: To enrich the agents' knowledge, we integrated Langchain's Wikipedia tools for external information retrieval. Wikipedia is an

immense, up-to-date knowledge source, and through Langchain's `WikipediaQueryRun` and `WikipediaAPIWrapper`, we can efficiently access its contents programmatically. The integration leverages Langchain's abstraction over the MediaWiki API, providing a streamlined way to search for and retrieve article summaries.

In practice, when a user asks a question that requires factual data not in the LLM's training (e.g., "Who is the president of country X as of this year?"), the AI agent utilizes our `WikipediaSource` class, which wraps Langchain's Wikipedia tools. This implementation allows for both targeted searches and summary retrievals through methods like `search()`, `get_summary()`, and `get_content()`. The retrieved information is then incorporated into the LLM's prompt through our knowledge enhancement pipeline.

The rationale for using Langchain's Wikipedia integration is twofold: it reduces hallucinations by providing current knowledge (as the LLM's internal knowledge might be outdated), and it leverages Langchain's robust error handling and rate limiting features. While this creates a dependency on both Langchain and Wikipedia's services, the asynchronous implementation ensures minimal impact on response times. To maintain efficient operation, we implemented limits on both the number of Wikipedia articles retrieved and the text length pulled, preventing the LLM's prompt from becoming unwieldy.

Multi-Agent Design and Benefits: The decision to structure the chatbot as multiple specialized agents was motivated by the complexity of user queries and the principle of divide-and-conquer. Multi-agent architectures allow each agent to be optimized for a subset of tasks or knowledge domains (LangGraph: Multi-Agent Workflows). In our design, the General Q&A agent has an open-domain prompt (more creative, broad coverage), the Admissions agent has a factual prompt with a restricted scope (focused on Concordia's programs and requirements), and the AI Expert agent uses a technical prompt and possibly a different style of answering (more detailed and jargon-friendly for AI topics). This specialization yields better results because an agent with a focused task can use a tailored prompt with appropriate constraints or examples, improving its reliability. As noted in LangChain's docs, *"Grouping tools/responsibilities can give better results. An agent is more likely to succeed on a focused task than if it has to select from dozens of tools"* (LangGraph: Multi-Agent Workflows). Similarly, we could even assign different underlying models if needed – e.g., a smaller, faster model for straightforward FAQ answers and a larger one for complex AI questions – though in our current setup all agents use the same base LLM. Another benefit is

modularity in development: each agent was implemented and tested independently with its own logic. This made debugging easier, as we could isolate issues within one agent's chain. The agents communicate primarily through the orchestrator, rather than directly with each other (a **Supervisor** architecture as per LangChain's terminology). The orchestrator acts as the single point that each agent reports to or receives tasks from, simplifying coordination. In future, this could be extended so that agents could call one another or work in sequence if a query spans multiple domains, but that adds complexity. We settled on a straightforward approach where one query is handled by one agent at a time, which covers our use cases. The trade-off of multi-agent systems is overhead in deciding which agent to use and potentially combining outputs, but our routing strategy (discussed below) was designed to be efficient and mostly deterministic based on query content.

Multi-Turn Context Management

Handling multi-turn conversations is implemented through a combination of custom conversation management and knowledge enhancement. The system uses a `ConversationManager` class to track and maintain conversation history, with each conversation having a unique ID and storing messages in a role-based format (user/assistant).

For context persistence, we use a simple but effective `MessageStore` class that maintains conversation history as a list of `BaseMessage` objects (from Langchain's core messages). The system is configured with a `MAX_HISTORY_LENGTH` setting (default: 10 turns) to prevent conversations from growing too large.

Knowledge enhancement is handled by our `KnowledgeEnhancer` class, which combines two types of knowledge sources:

1. Vector store (using FAISS/Chroma) for retrieving relevant information from our knowledge base
2. Wikipedia integration through Langchain's Wikipedia tools

When processing queries, the system:

1. Maintains conversation context through the `ConversationManager`
2. Enhances queries with relevant knowledge using `KnowledgeEnhancer`

3. Formats both conversation history and retrieved knowledge into the prompt template
4. Uses a `ChatPromptTemplate` that includes system context, conversation history, and relevant knowledge

The prompt structure in our `BaseAgent` class looks like:

```
self.prompt = ChatPromptTemplate.from_messages([
    ("system", "You are a specialized assistant named {name}. {description}"),
    MessagesPlaceholder(variable_name="history"),
    ("system", "Relevant Information:\\n{knowledge}"),
    ("human", "{input}")
])
```

This implementation provides a clean separation of concerns between conversation management, knowledge enhancement, and agent processing, while maintaining context effectively across multiple turns.

Another consideration was **agent-specific memory** vs **global memory**. We debated if each agent should have its own vector index (storing only domain-specific interactions) or if all conversations should go into one global index. We chose to use a single global index but with metadata tagging. Each vector (message embedding) is tagged with which agent produced it or if it was a user message. This way, when retrieving, we can filter by agent or decide to retrieve across all. In practice, general conversations largely stay in one agent, so a global memory works and allows cross-agent context carrying. For example, if a user first asks a general question and then asks something that gets routed to the AI agent, the AI agent can still see the general conversation history from before. The risk is that irrelevant stuff might surface, but our similarity search ensures only semantically related history is pulled.

Routing Logic for the Orchestrator

Designing the agent selection (routing) mechanism was implemented with a straightforward and efficient approach using a rule-based keyword matching system. The routing logic is implemented in the `MultiAgentCoordinator` class, which

uses predefined keyword sets to determine which specialized agent should handle a query.

The implementation uses two main keyword sets:

1. Concordia CS keywords: Terms like "concordia", "university", "admission", "computer science", "cs program", "application", "requirements", "gpa", "deadline", "tuition", etc.
2. AI-related keywords: Terms like "artificial intelligence", "machine learning", "deep learning", "neural network", "nlp", "computer vision", "reinforcement learning", etc.

The routing mechanism works through a simple but effective counting system:

1. First, it converts the query to lowercase for case-insensitive matching
2. Then counts matches for each category in the current query
3. Routes to the specialized agent with the highest number of keyword matches
4. Defaults to the General Agent if no clear category is detected

A key feature of our implementation is context-aware routing. If the current query doesn't have clear keyword matches, the system examines recent conversation history (up to 4 messages) to maintain context. For example, if a user previously asked about Concordia admissions and follows up with "What about the GPA?", the system will maintain routing to the Concordia CS Agent even though the follow-up query alone doesn't contain explicit Concordia keywords.

The system is configured through centralized settings, with keyword sets defined in the coordinator class for easy maintenance. While this approach is simple, it has proven effective for our use case, providing fast and predictable routing without the computational overhead of LLM-based routing solutions.

Future improvements could include:

- Expanding keyword sets based on actual usage patterns
- Adding regex patterns for more complex matching
- Implementing confidence scores for routing decisions
- Adding support for multiple agent collaboration on complex queries

Once the MultiAgentCoordinator selects an agent through its keyword-based routing system, it processes the query through a well-defined pipeline:

1. First, it manages conversation state:
    - Creates a new conversation if needed or retrieves existing one
    - Adds the user's message to conversation history through ConversationManager
    - Retrieves relevant conversation history formatted for the agent
2. Then, it enhances the query with external knowledge:
    - The KnowledgeEnhancer retrieves relevant information based on the agent's configuration
    - For the General Agent: Uses Langchain's Wikipedia tools
    - For the AI Agent: Combines both Wikipedia and vector store results
    - For the Concordia CS Agent: Uses only vector store results
    - Formats the retrieved knowledge into a structured prompt format
3. The selected agent then processes the query through its chain:
    - Uses a ChatPromptTemplate that combines:
        - System context (agent name and description)
        - Conversation history (through MessagesPlaceholder)
        - Retrieved knowledge context
        - User's input
    - Invokes the Ollama LLM with the prepared prompt
    - Adds the interaction to its MessageStore for future context
4. Finally, the coordinator:
    - Adds the agent's response to the conversation history
    - Returns a response object containing:
        - The agent's response

- The agent type used
- The conversation ID

This entire process is implemented asynchronously, leveraging FastAPI's async capabilities. While the multi-agent architecture adds some complexity, the modular design with clear separation of concerns (routing, knowledge enhancement, conversation management, and agent processing) makes the system maintainable and extensible. The overhead from agent selection is minimal since it uses efficient keyword matching rather than LLM-based routing.

Code Organization and Implementation Details

Our codebase is organized with a clear modular structure under the `demo` directory:

```
demo/
├── docs/ # Documentation files
├── src/ # Source code
│ ├── agents/ # Agent implementations
│ ├── api/ # FastAPI endpoints
│ ├── config/ # Configuration management
│ ├── knowledge/ # Knowledge base integration
│ └── utils/ # Utility functions
├──
main.py # Application entry point
└──
todo.md # Development tasks
```

Key components and their implementations:

- `src/agents/` – Contains our agent implementations:
  - `base_agent.py`: Abstract base class with shared functionality
  - `general_agent.py`: Handles general queries with Wikipedia integration
  - `concordia_cs_agent.py`: Specializes in CS admissions with vector store
  - `ai_agent.py`: AI-focused agent using both Wikipedia and vector store
  - `coordinator.py`: Implements the MultiAgentCoordinator for query routing

- `src/api/` – FastAPI implementation:
  - `app.py`: FastAPI application factory with CORS and router setup
  - `router.py`: API endpoint definitions for chat and agent listing
  - Async endpoints that integrate with our coordinator
- `src/knowledge/` – Knowledge integration:
  - `wikipedia_source.py`: Langchain's Wikipedia tools integration
  - `vector_store.py`: FAISS/Chroma vector store implementation
  - `enhancer.py`: KnowledgeEnhancer for combining knowledge sources
- `src/config/` – Configuration management:
  - `config.py`: Central configuration for Ollama, agents, and knowledge sources
  - Environment-based settings (OLLAMA_BASE_URL, API settings, etc.)
  - Agent definitions and knowledge source configurations
- `src/utils/` – Utility functions:
  - `conversation.py`: ConversationManager for multi-turn chat handling
  - Other utility functions and helpers

The system's key architectural features:

1. **Agent System**:
   - Each agent inherits from BaseAgent
   - Custom MessageStore for conversation tracking
   - Agent-specific knowledge source configurations
   - Async query processing with Ollama integration
2. **Routing Logic**:
   - Keyword-based routing in MultiAgentCoordinator
   - Context-aware routing using conversation history
   - Efficient domain classification through keyword matching
3. **Knowledge Enhancement**:

- Langchain's Wikipedia tools for external knowledge
- FAISS/Chroma vector store for document retrieval
- Agent-specific knowledge source combinations

4. **Conversation Management**:
    - Custom ConversationManager for session handling
    - Configurable history length (MAX_HISTORY_LENGTH)
    - Clean separation of conversation state and processing

The codebase leverages async/await patterns throughout, from the FastAPI endpoints to the agent query processing. Error handling is implemented at various levels, with proper exception handling in the Wikipedia integration and vector store operations.

Configuration is centralized and environment-based, making it easy to adjust settings like the Ollama model, API endpoints, and knowledge source configurations. The modular design allows for easy extension of functionality, such as adding new specialized agents or knowledge sources.

Implementation Challenges

Our development of the adaptive multi-agent chatbot system encountered several significant challenges, primarily centered around local LLM integration, agent coordination, and knowledge management.

Core Technical Challenges

The foundation of our system relies on local LLM execution through Ollama, which presented our first major challenge. We implemented a straightforward integration using environment variables (OLLAMA_BASE_URL, OLLAMA_MODEL) for configuration, but this basic setup means we're limited to CPU-only execution with standard model settings. This directly impacts response times and resource utilization, as we're constrained by local hardware capabilities without any advanced optimization strategies.

Agent coordination proved to be another complex challenge. Our solution implements a keyword-based routing system through the MultiAgentCoordinator,

where each agent type (General, CS Admissions, and AI) has specific knowledge source configurations. The General Agent leverages Wikipedia integration, the AI Agent combines Wikipedia with vector store access, and the CS Agent exclusively uses the vector store. While functional, this approach has limitations - it relies on simple keyword matching without sophisticated NLP, and the domain boundaries are relatively rigid.

Knowledge integration presented its own set of challenges. We developed a system that combines Wikipedia access through Langchain's tools with a FAISS vector store for document retrieval, managed through our custom KnowledgeEnhancer class. Each agent can be configured to use different knowledge sources, but the implementation remains basic - there's no caching layer for repeated queries, and we're limited in the types of knowledge sources we can integrate.

System Management Challenges

Conversation state management required careful consideration. We implemented a custom ConversationManager class that handles session management with a configurable history length (set to 10 turns). While this provides basic conversation tracking, it's limited by its in-memory storage nature - conversations don't persist between server restarts, and the history trimming mechanism is fairly basic.

The asynchronous nature of our system presented another layer of complexity. While we successfully implemented async patterns throughout the codebase using FastAPI and async methods in our agent implementations, our implementation focuses on basic async/await patterns without advanced concurrency handling or sophisticated error recovery strategies.

Infrastructure and Development Gaps

Our current implementation has several notable gaps that impact its robustness and maintainability. The most significant is the complete absence of a testing infrastructure - we have no unit tests, integration tests, or testing framework in place. This makes it difficult to verify system behavior and catch potential regressions.

Error handling in the system is minimal, with only basic error handling in API routes and limited error recovery strategies. We also lack any systematic logging or monitoring infrastructure, making it difficult to track system performance and identify issues in production.

The vector store implementation, while functional, is relatively basic. We're using FAISS for document embedding and retrieval, but without advanced indexing strategies or optimizations for large-scale operations. This could become a bottleneck as the knowledge base grows.

Configuration and Documentation State

Our configuration management is centralized through config.py and environment variables, providing a workable but basic solution for managing system settings. Documentation exists primarily through OpenAPI-generated API documentation and basic internal code comments, but lacks comprehensive deployment guides or detailed internal API documentation.

Looking Forward

These challenges point to several clear areas for improvement:

- Development of a comprehensive testing infrastructure
- Implementation of robust error handling and logging systems
- Addition of caching and performance optimizations
- Enhancement of the vector store implementation with more sophisticated features
- Expansion of system documentation and monitoring capabilities

While our current implementation provides a functional multi-agent chatbot system, these limitations represent opportunities for future development and enhancement rather than insurmountable obstacles. The modular architecture we've established provides a solid foundation for implementing these improvements incrementally.

Future Considerations

While our multi-agent chatbot system is fully functional, there are several **future improvements and considerations** that can enhance its performance, scalability, and maintainability:

- **GPU-Based Optimization:** Deploying the LLM on a GPU would dramatically speed up inference. Currently, if running on CPU, the response time and throughput are limited. In the future, we plan to use a machine with an NVIDIA GPU to host the Ollama model (Ollama supports GPU acceleration, automatically using it if available (Running LLMs locally with Ollama. Introduction | by Omar Alva | Medium)). This would allow us to use a larger model for better answer quality without sacrificing responsiveness. Another avenue is exploring model distillation or quantization techniques further – perhaps running two models: a smaller one for routing and trivial queries and a larger one for complex queries. On the vector database side, if the data grows, using FAISS's GPU index capabilities could make vector searches faster as well, though currently that's not a bottleneck.
- **Improved Scalability:** To handle more users or higher query volumes, we need to scale the system horizontally. FastAPI can be run with multiple workers (using Uvicorn/Gunicorn), but since we have a stateful memory (FAISS index in memory), we'd need to ensure each worker has access to the same memory or use a centralized vector store. In future, we might extract the memory component into a separate service (or use a networked vector database like Pinecone or Weaviate) so that multiple app instances can share context. Another improvement is streaming responses: currently the user waits for the full answer. We could utilize FastAPI's StreamingResponse and Ollama's streaming API to send partial results back as they are generated, improving the interactive feel. Monitoring and load balancing will also be crucial — we'd integrate an APM (Application Performance Monitoring) tool to watch latency, and possibly add a rate limiter or queue if too many requests come in simultaneously for the single-model backend to handle.
- **Enhanced Monitoring and Logging:** In a production scenario, we want better observability. We plan to add more robust logging of user interactions (while respecting privacy – possibly logging only metadata). Monitoring the health of the Ollama process is also important; if it crashes or becomes unresponsive, the system should detect that and restart it or return a graceful error. We could

integrate heartbeat checks or use Ollama's built-in endpoints to verify it's loaded a model. Additionally, tracking the usage of each agent (how often each is invoked) can guide future training – e.g., if the general agent is handling the majority, maybe we need to improve our routing or train the other agents on more data.

- **Security Enhancements:** As an API available to end users, we must consider security. One concern is **prompt injection** – a user could try to trick the system by including malicious instructions in their input (like "ignore previous instructions"). Although each agent's prompt tries to guard against this (we instruct the model not to reveal system prompts or to stay in its role), adaptive adversarial prompts are an ongoing cat-and-mouse issue. We will explore using an additional filtering layer: for instance, scanning user inputs for certain keywords or patterns and sanitizing them (especially for the Wikipedia tool – e.g., ensure we don't pass a query that breaks the Wikipedia API or causes an unintended request). Another security aspect is preventing misuse of the system; since it can execute code (in theory if we added such tools) or access external info, we limit the tools and ensure agents cannot call arbitrary Python functions. FastAPI itself needs to be secured – we could add authentication or an API key requirement if this were public. We also consider data security: conversations could be sensitive, so we plan to add optional encryption for stored vectors on disk and in memory (at least ensure that if the server is shut down, the conversation history is not trivially accessible). These measures will become more important if the system is deployed to a cloud or shared environment.
- **Extending Knowledge Integration:** Currently, Wikipedia is the only external knowledge source, and the Admissions agent relies on a pre-loaded small dataset. In the future, we envision integrating more sources. For Concordia admissions, we could connect an API or scrape the official website periodically and update the vector store so that the latest information is used. For AI questions, maybe connect to arXiv or other scientific knowledge bases for up-to-date research information. These would each be new tools or retrievers the respective agent can call. With multiple tools, we might explore a more advanced agent planning mechanism (like LangChain's ReAct framework) where an agent can decide a sequence of actions (e.g., search

Wikipedia, then use that info to form an answer). That would make the agents more autonomous in figuring out when to use which resource.

- **Learning and Adaptation:** Since this is an "adaptive" system, a future direction is to incorporate learning from interactions. We could store conversation logs and have a process to fine-tune the model or adjust prompts based on where it failed. For example, if users often rephrase questions indicating the answer was unsatisfactory, that's a signal. Fine-tuning a local model with project-specific Q&A pairs (especially for the admissions domain) could make the model itself an expert, reducing reliance on vector retrieval. We could also experiment with feedback loops: allow users to mark answers as helpful or not, and use that to weight the vector memory (e.g., prefer using info from interactions that were rated highly).
- **User Interface and Deployment:** Though not the core of this report, it's worth noting that we plan to build a simple web frontend (perhaps using Streamlit or a React app) that interfaces with the FastAPI backend. This would make the chatbot more accessible to users who can then chat with it in a browser. Deployment-wise, containerizing everything (which we partially did) and maybe using a cloud service or a VM to host it would allow us to share the chatbot for broader testing. If usage grows, we'd consider orchestrating it with Kubernetes, which would also make scaling and updating easier.

In conclusion, there are many opportunities to enhance the system. Our immediate focus will be on performance (adopting GPUs) and robustness (better testing and security), which will provide a solid foundation for any further feature expansions. We view this project as a continuous learning platform — as LLM technology and frameworks evolve, we plan to incorporate the latest techniques (for instance, new memory architectures or agent frameworks) to keep the system up-to-date and improve its capabilities.

Conclusion

This project succeeded in creating a comprehensive multi-agent chatbot that demonstrates the power of combining local AI models with specialized agent design. We began with the goal of answering a diverse set of user queries ranging from general trivia to university admissions and advanced AI topics. By decomposing the problem into multiple agents, we achieved more focused and

accurate responses for each domain, as each agent could be optimized with its own prompt and knowledge. The use of **LangChain** facilitated the complex orchestration of these agents and provided memory mechanisms that gave our chatbot a conversational feel, remembering what was said earlier and who the user is (to the extent configured).

One of the key strengths of our system is its emphasis on **privacy and autonomy**: running the LLM via Ollama locally means users (or institutions deploying it) are not dependent on external APIs and do not have to send potentially sensitive data to third parties. The incorporation of **FAISS** for memory made the chatbot capable of referencing past dialogue and external documents efficiently, which is a notable improvement over stateless question-answering systems. Additionally, the **modular architecture** means the system is extendable — new agents or tools can be added with relative ease. For example, adding an agent for math problem solving or for a different university's admissions could reuse much of the existing framework.

However, the project also highlighted areas for improvement. **Performance** is a concern, as local inference is slower than cloud-based alternatives; scaling up will require hardware upgrades or optimizations like model quantization or switching to more powerful model versions on GPU. The **agent routing**, while working, could be made more sophisticated to handle ambiguous queries better — occasionally the system might choose a non-ideal agent if the query is on the borderline of two domains. In terms of knowledge, the chatbot is only as good as the information we give it: keeping the admissions data updated and expanding the AI agent's knowledge base (perhaps integrating with an AI news feed) will be necessary to maintain its usefulness over time. We also recognize the need for a more thorough **evaluation** of the system's responses: an academic report would typically evaluate correctness and user satisfaction, which we did informally but could be done more rigorously with user studies or benchmark tests.

From a software engineering perspective, the project underscored the importance of balancing innovation with reliability. We integrated cutting-edge tools and models, but also had to ensure the system remains stable and debuggable. We managed to create a codebase that a future developer can navigate (with clear separation of components and extensive comments), though there is always some technical debt in early prototypes. Moving forward, we are optimistic that with iterative enhancements the Adaptive Multi-Agent Chatbot can become even more

responsive, intelligent, and useful for users. It serves as a solid foundation for any application that may require a **conversational AI with expertise in multiple areas**, demonstrating how such a system can be built with current open-source technologies.

In conclusion, this report documented the journey of designing an advanced chatbot system at an undergraduate project level — from conceptual architecture to concrete implementation details and challenges. The result is a functional system that showcases a combination of **local LLM deployment**, **multi-agent collaboration**, and **retrieval-based augmentation** to deliver a rich interactive experience. The project not only meets its initial requirements but also opens the door to many exciting future directions where AI agents can adapt and work together to assist users more effectively.

References

- Harrison Chase *et al.*, **LangChain** – *"LangChain is a software framework that helps facilitate the integration of large language models (LLMs) into applications."* – LangChain Documentation (LangChain - Wikipedia)
- **FastAPI Framework** – *"FastAPI is a modern, fast (high-performance), web framework for building APIs with Python based on standard Python type hints."* – FastAPI Documentation (FastAPI)
- **Ollama – Local LLM Engine** – Omar Alva, *"Ollama is a user-friendly interface for running large language models (LLMs) locally… supports a wide range of models, including Llama 2…"* – *Medium* (Feb 2024) (Running LLMs locally with Ollama. Introduction | by Omar Alva | Medium)
- **Local LLM Advantages** – *"Privacy concerns? Gone! … No more waiting for API calls … Run as many inferences as your hardware can handle."* – Dev.to on running LLMs locally (Ollama Cheatsheet: Running LLMs Locally with Ollama - DEV Community)
- **FAISS (Facebook AI Similarity Search)** – *"Faiss is a library for efficient similarity search and clustering of dense vectors. It contains algorithms that search in sets of vectors of any size…"* – Faiss Documentation (Welcome to Faiss Documentation — Faiss documentation)

- **LangChain Memory (Vector Store Retriever)** – *"When added to an agent, the memory object can save pertinent information from conversations or used tools."* – LangChain Docs on VectorStoreRetrieverMemory (Backed by a Vector Store | LangChain)
- **LangChain Multi-Agent Design** – *"Multi-agent designs allow you to divide complicated problems into tractable units of work that can be targeted by specialized agents and LLM programs."* – LangChain Blog (LangGraph) (LangGraph: Multi-Agent Workflows)
- **MediaWiki/Wikipedia API** – *"The MediaWiki Action API is a web service that allows access to some wiki features like authentication, page operations, and search."* – MediaWiki API Documentation (API:Main page - MediaWiki)
- **Wikipedia Integration in LangChain** – *"Wikipedia is a multilingual free online encyclopedia... This notebook shows how to retrieve wiki pages from wikipedia.org into the Document format used downstream."* – LangChain Wikipedia Retriever Docs (WikipediaRetriever | LangChain)