Technical Project Report: Development of the UCY AI Assistant – A Retrieval Augmented Generation System for the University of Cyprus

Project Team: Mikhail Sumskoi, Marios Menelaou, Maria Tsilidou, Chrysis Andreou, Constantinos Leonidou

Table of Contents

---

1. Introduction

This report details the development of the University of Cyprus (UCY) AI Assistant, a Retrieval Augmented Generation (RAG) system designed to enhance information accessibility for students, faculty, and staff. In an academic environment, navigating the vast and often complex web of university-related information can be challenging. The UCY AI Assistant aims to address this by providing a conversational interface capable of understanding user queries and delivering accurate, contextually relevant answers derived from official UCY data sources.

The primary objective of this project was to explore and implement RAG technology to solve the information retrieval problem at UCY. The team recognized that a multifaceted approach would be most beneficial. This involved an initial phase focused on building a RAG system from fundamental principles to gain a deep understanding of its core mechanics. This foundational work served as an essential learning experience. Following this, the team broadened its exploration to encompass production-grade libraries and frameworks, which offer more robust, scalable, and feature-rich solutions suitable for a deployable system. The team's exploration included investigating the Langchain and LangGraph frameworks, as well as the OpenAI Agents SDK.

This report will narrate the project's journey, starting with the insights gained from the low-level implementation and then detailing the subsequent explorations into higher-level frameworks. It will compare these distinct approaches, analyze their respective strengths and weaknesses, and outline the team's rationale for this phased strategy. Finally, it will propose next steps for the continued development of the UCY AI Assistant.

2. Foundational RAG Implementation: A Deep Dive Approach

To establish a comprehensive understanding of the intricacies involved in Retrieval Augmented Generation, an initial phase of the project focused on developing a RAG system from the ground up. This effort, detailed in a "report" for the "RagDeepThought Project," was undertaken to meticulously explore

and implement each component of the RAG pipeline, thereby providing the team with invaluable insights into the core mechanics.

2.1. Data Preprocessing Pipeline The team's approach began with a multi-stage data preprocessing pipeline designed to curate a high-quality corpus from UCY web data. As documented in the related report (RagDeepThought), the process involved:

1. **Data Acquisition:** A list of UCY-related URLs (`01_links.csv`) was used to fetch HTML content, employing proxies to manage potential IP blocking.
2. **Format Conversion:** The raw HTML was converted to Markdown format using `markdownify` for easier text manipulation.
3. **Content Cleaning:** A significant cleaning step was applied to the Markdown files. This involved identifying the main content block (typically starting after the first H1 header) and attempting to remove common boilerplate elements like navigation menus and footers using regex patterns. Whitespace was also normalized.
4. **Filtering:** Finally, documents were filtered based on length (minimum 100 words) to ensure only substantially informative pages were retained, resulting in a directory of `long_files/` ready for RAG ingestion.

This systematic preprocessing was crucial for ensuring the quality and relevance of the documents fed into the RAG system.

2.2. Core RAG Architecture The core RAG architecture in this implementation (`RagDeepThought_pre-run` , `rag_core.py`) was built with a focus on understanding each step:

- **Data Ingestion and Chunking:** Cleaned Markdown documents were loaded and further processed. Text was chunked using `SentenceSplitter` from `llama_index.core.node_parser` with a configured `CHUNK_SIZE` of 200 characters and an overlap of 20 characters. This strategy aimed for concise, semantically coherent chunks.
- **Embedding:** Document chunks were embedded using the `intfloat/multilingual-e5-large-instruct` model, chosen for its strong performance in semantic retrieval and multilingual capabilities, pertinent to UCY's bilingual context. Embeddings were L2-normalized.
- **Vector Indexing:** A FAISS `IndexFlatIP` (Inner Product) index was built from the normalized chunk embeddings. For normalized vectors, inner product is equivalent to cosine similarity, facilitating efficient and accurate semantic search. The index (`indexE5.faiss`) contained embeddings for 1002 chunks.
- **Retrieval:** Given a user query, it was embedded using the same model, normalized, and then used to search the FAISS index for the top-K most similar chunks.
- **Answer Generation:** The retrieved chunks were passed as context, along with the original question, to a Large Language Model (LLM). The `meta-llama/Llama-3.3-70B-Instruct-Turbo-Free` model, accessed via the Together AI API, was used for generating the final answer. Prompt engineering was employed to guide the LLM to synthesize answers based strictly on the provided context.

2.3. Synthetic Question-Answer (QA) Dataset Generation A significant innovation in this work was the methodology for generating a synthetic QA dataset for evaluation (`1_dataset_pipeline.py`). This process was designed to create relevant and diverse evaluation data directly from the UCY corpus:

1. **Representative Chunk Selection:** To ensure broad topic coverage, k-means clustering was performed on all 1002 document chunk embeddings. For 1002 chunks, this resulted in `k=31`

clusters.
2. **Sampling from Clusters:** A target of 42 QA pairs was set (a thematic choice linked to the "DeepThought" persona of the project). Chunks were sampled proportionally from each cluster, with the chunk closest to the centroid selected first, followed by diversity sampling for additional chunks from the same cluster. This ensured that the selected chunks for QA generation were varied.
3. **LLM-based QA Pair Generation:** The selected 42 chunks were then fed to the `meta-llama/Llama-3.3-70B-Instruct-Turbo-Free` LLM, prompted to generate a question and a corresponding answer based solely on the provided chunk text.
4. **Dataset Augmentation:** Each generated QA pair was augmented by identifying the top-5 most semantically similar chunks from the entire corpus to its source chunk, which served as ground-truth relevant documents for retrieval evaluation. The final dataset was saved as `generated_qa_dataset_ranked.xlsx`.

2.4. Evaluation Methodology and Results The evaluation framework (`3_evaluate_retrieval.py`, `4_evaluate_generation.py`) assessed both retrieval and generation components using the synthetic QA dataset.

- **Retrieval Metrics:** Hit Rate@K, Mean Reciprocal Rank (MRR)@K, Precision@K, and Kendall's Tau-b@K were calculated for K values of 1, 3, and 5.
- **Generation Metric:** Mean Semantic Similarity (cosine similarity between embeddings of generated and ground-truth answers) was the primary metric.

The evaluation was conducted on two versions of the QA dataset: one with all automatically generated questions and another manually curated "good questions" set (33 Qs) where poorly formed or non-specific questions were removed. Results highlighted the significant impact of question quality:

- **With "Good Questions" (vs. Auto-Generated):**
  - Mean Hit Rate @5 improved from 0.7381 to 0.8485.
  - MRR @5 improved from 0.5877 to 0.6490.
  - Mean Semantic Similarity (generation) improved from 0.9429 to 0.9631.

These findings emphasized that clearer, more specific questions lead to better retrieval and, consequently, higher quality generated answers.

This foundational work provided a granular understanding of the RAG pipeline, establishing the critical role of data quality, chunking strategy, embedding model choice, and evaluation methodologies. This deep dive was instrumental for the team before progressing to more abstracted, production-oriented frameworks.

3. High-Level Framework Exploration: Langchain-Based Agent

Following the foundational exploration, the team developed a more sophisticated RAG agent using high-level frameworks, specifically Langchain and LangGraph. This work aimed to build a modular and extensible UCY AI assistant.

3.1. Implementation using Langchain and LangGraph The team leveraged Langchain for its comprehensive suite of tools for building LLM applications and LangGraph for creating robust, stateful agentic systems. LangGraph allowed for the definition of a cyclical graph where nodes represent computational steps (e.g., calling an LLM or a tool) and edges represent the flow of logic. This approach enabled the construction of a multi-tool agent capable of complex decision-making.

3.2. Architecture: Agentic System with Specialized Tool Nodes The core of this implementation was an agentic system orchestrated by LangGraph. The agent was designed to understand user queries and route them to one of several specialized tool nodes, each responsible for handling a specific type of information or task. As described in the MAI623 Report-2, these tools included:

- **General Question Node:** This node addressed general user questions regarding university courses, fees, program content, etc. It drew information from a Pinecone index populated with embeddings from web-scraped UCY URLs and general university documentation (PDFs, course brochures). This node served as the backbone of the chatbot.
- **Schedule Question Node:** This tool allowed users to fetch specific details about course schedules, such as room numbers, instructor names, buildings, and lesson times. It queried a dedicated Pinecone index containing information extracted from schedule PDF files (converted to JSON).
- **Form Node:** This node retrieved URLs for various UCY forms. It utilized a Pinecone index containing descriptions and links to these forms, enabling users to find the correct form based on their needs.

The LangGraph agent would receive a user query, and the LLM (OpenAI `gpt-4o`) would decide which tool (or tools) to invoke. After a tool executed and returned its results, the information was passed back to the LLM to synthesize a final answer. The system also incorporated `MemorySaver` for maintaining conversation history, allowing for contextual follow-up interactions.

3.3. Data Sources and Preprocessing The agent relied on a diverse set of data sources, meticulously prepared and ingested into Pinecone vector stores:

- **URLs:** UCY website content was scraped using ParseHub and Spider Web Crawler. The extracted text was processed and chunked.
- **PDFs:** General university documents (brochures, guides) were processed. Work was done on extracting text from PDFs and DOCX files, pre-processing it (cleaning, lowercasing, removing boilerplate, fixing hyphenation), and generating keyword-rich descriptions using an LLM. These descriptions, merged with metadata like filenames and links, were vital for enhancing retrieval.
- **JSON Schedules:** Course schedules, originally in PDF format, were converted to JSON objects using Mistral OCR. Then, the JSON schedule files were parsed and prepared for embedding.
- **Forms Data:** Information about UCY forms, including names, links, and descriptions (LLM-generated by the team), was structured, as JSON objects, and embedded.

Chunking Strategy: The team adopted a strategy of creating relatively large chunks (e.g., 1500-1700 characters with a 500-character overlap using Langchain's `RecursiveCharacterTextSplitter`) to ensure that general ideas were retained within embeddings.

Embedding Model and Vector Store:

- **Embedding Model:** `VoyageAI voyage-3-lite` with 1024 dimensions was chosen for its strong multilingual performance and ability to handle both English and Greek, a crucial requirement for UCY data.
- **Vector Store:** Pinecone was used as the vector database. Separate Pinecone indexes were set up for different data types (general info, URLs, schedules, forms) to support the specialized tool nodes.

3.4. Evaluation Strategy and Reported Metrics The team employed a multi-faceted evaluation approach:

- A synthetic dataset of factoid questions was generated by feeding shuffled LangChain documents (and their metadata) to an LLM.
- **Retrieval Evaluation:** Metrics such as Hit Rate @ k (e.g., 0.76 for URL data, 0.88 for Schedule data) and Precision @ k (e.g., 0.59 for URL data) were used. The Hit Rate was defined as whether the target document was present among the top-k retrieved documents.
- **Generation/Overall Evaluation:** An "llm-as-a-judge" approach was used, where an LLM evaluated the quality and correctness of the RAG system's final answers. The MAI623 Report-2 indicates high scores (e.g., 0.88 for URL data, 0.85 for Form Fetcher), with around 85-90% of responses deemed factually correct and containing the reference answer.

This Langchain-based agent demonstrated the power of high-level frameworks in rapidly developing a modular, tool-driven RAG system. The collaborative effort in data preparation and tool creation highlighted a practical approach to building a comprehensive AI assistant.

4. High-Level Framework Exploration: OpenAI Agents SDK Approach

Concurrently with the Langchain exploration, the OpenAI Agents SDK was investigated as an alternative high-level framework. This work focused on building a RAG agent with a declarative structure, emphasizing orchestration and state management for complex workflows.

4.1. Implementation using OpenAI Agents Framework The OpenAI Agents Framework was chosen for its capabilities in simplifying the coordination of sub-agents, task planning, seamless tool integration, and stateful context management. This framework allows for a more declarative definition of agent behaviors and interactions.

4.2. Architecture: Modular Design with Triage Agent and Sub-agents The architecture featured a modular design orchestrated by a master agent:

- **Triage Agent (Master Agent):** This central agent controlled the overall workflow, managing task delegation to specialized sub-agents and ensuring the correct sequence of operations. It was designed to support a "Human-in-the-Loop" pattern, allowing sub-agents to return control to the user for clarification if needed.
- **Sub-agents/Tools:** The Triage Agent would orchestrate calls to various tools or sub-agents. Two key tools/processes highlighted were:
  - `retrieve_general_documents`: This tool was responsible for fetching relevant document chunks from a vector store based on the user's query.
  - `grade_documents`: A crucial component for refining retrieved information. This tool used an LLM to assess the relevance of each retrieved chunk against the user's question. Irrelevant chunks were filtered out, enhancing the precision of the context provided for answer generation.
- **Short-Term Memory Management (`RagAgentContext`):** A Pydantic model, `RagAgentContext` (defined in `src/context_model.py`), was used to maintain the agent's state throughout a conversation. This context preserved the user's question, retrieved chunks, intermediate outputs (like the graded list of documents), and conversation history, enabling coherent multi-turn interactions and shared state between different parts of the workflow.

The orchestration flow typically involved the Triage Agent invoking the Retrieval Agent/tool, whose output could then be passed to a Grader Agent/tool before final answer generation.

4.3. Data Sources, Embedding, and Vector Store This implementation relied on a pre-built vector store:

- **Data Sources:** The agent used data from JSON files located in a `data/` directory, including `course_list.json`, `forms.json`, and `scraped_urls.json` (as seen in `warmup.ipynb`).
- **Embedding Strategy:** The `Ollama nomic-embed-text` model was used for generating embeddings. This was configured in the `warmup.ipynb` script and implicitly used by the `retrieve_general_documents` tool. This choice allowed for local embedding generation, assuming a running Ollama instance.
- **Vector Store:** FAISS was used as the vector store. An essential preliminary step, detailed in `warmup.ipynb`, involved loading the source documents, chunking them, generating embeddings with `nomic-embed-text`, and building a FAISS index. This index was saved locally in the `faiss_index/` directory and loaded at runtime by the `retrieve_general_documents` tool.

4.4. Key Techniques Several notable techniques were employed in this approach:

- **XML Tags for Prompt Engineering:** XML tags were used within prompts to structure information for the LLM. This method helps in clearly delimiting system instructions, retrieved context, and user queries, which can improve the LLM's instruction-following accuracy and focus.
- **Pydantic Schemas for Structured Output:** For tools like `grade_documents`, Pydantic schemas (e.g., `Grade`, `DocumentGrade` in `src/tools.py`) were used to define the expected structure of the LLM's output. This enforces a reliable format for the relevance grades, making it easier to parse and use programmatically.
- **LLM-based Document Relevance Grading:** The `grade_documents` tool exemplifies using an LLM not just for generation but also as an intelligent component within the retrieval pipeline to filter and refine information.

This exploration of the OpenAI Agents SDK showcased a powerful approach to building complex, stateful RAG agents with a focus on declarative orchestration and advanced techniques like LLM-based document grading and structured prompting.

5. Team Rationale and Project Narrative

The overarching strategy for the UCY AI Assistant project, guided by discussions with the professor, was to undertake a comprehensive learning journey. This journey began with foundational principles and progressively advanced towards exploring sophisticated, production-ready solutions.

The initial decision to develop a low-level RAG system (the "RagDeepThought Project") was deliberate. The primary rationale was to ensure the team gained a fundamental, hands-on understanding of every component within a RAG pipeline. By building data preprocessors, chunking mechanisms, embedding strategies, vector indexing, retrieval logic, and generation prompts from scratch, the team could dissect the "black box" often associated with higher-level libraries. This foundational phase was crucial for appreciating the nuances, challenges, and trade-offs inherent in RAG systems, such as the impact of chunk size, the importance of embedding quality, the mechanics of vector search, and the subtleties of prompt engineering for both QA dataset generation and final answer synthesis. The detailed report (RagDeepThought) and systematic evaluation from this initial phase, particularly the insights on how question quality affects performance, provided a solid empirical basis for the team.

Having established this core understanding, the project's narrative then shifted towards exploring how these fundamental concepts are implemented and abstracted within production-grade libraries and frameworks. This led to the parallel explorations using Langchain/LangGraph and the OpenAI Agents SDK. The objective here was to evaluate tools that could accelerate development, offer greater scalability, and provide more advanced features like agentic behavior, tool use, and state management.

The Langchain exploration focused on its rich ecosystem, demonstrating how to build a multi-tool agent capable of handling diverse query types by routing them to specialized retrievers. This approach highlighted rapid development capabilities and the benefits of a modular, tool-based architecture. The work on data processing and description generation, and on schedule data, fed into this, showcasing a collaborative team effort in populating the knowledge base for such an agent.

Simultaneously, the work with the OpenAI Agents SDK explored a different paradigm for agent orchestration, emphasizing declarative structures, advanced state management with `RagAgentContext`, and sophisticated sub-agent/tool interactions like LLM-based document grading. This provided insights into building more complex and robust agentic systems where fine-grained control over agent behavior and inter-agent communication is paramount.

A key aspect of this multi-pronged approach was the implicit understanding that direct quantitative comparison between the three systems would be challenging. The initial system used specific datasets (UCY web data) and evaluation metrics tailored to its foundational goals. The Langchain/LangGraph exploration used a broader range of data sources (URLs, PDFs, JSON schedules) and a different synthetic QA generation and evaluation approach (llm-as-a-judge, Hit Rate@k on their dataset, as detailed in MAI623 Report-2). The OpenAI Agents SDK exploration focused more on the architectural implementation and advanced features like document grading. This divergence in datasets, specific model choices (embeddings, LLMs), and evaluation methodologies was a natural consequence of explorations aiming for different facets of RAG development rather than a head-to-head bake-off on a single benchmark. The value lay in understanding the qualitative differences, development experiences, and unique capabilities offered by each approach and framework.

This phased strategy—from deep, low-level understanding to broad, high-level framework exploration—equipped the team with a holistic perspective on RAG technology, preparing them for informed decisions about the future architecture of the UCY AI Assistant.

6. Comparative Analysis of the Three RAG Approaches

The project's exploration yielded three distinct RAG implementations, each offering unique insights into building information retrieval systems. A comparative analysis highlights their differences in methodology, technical stack, targeted functionalities, and overall strengths and weaknesses.

6.1. Low-Level RAG Implementation

- **Methodology:** Built from fundamental components, emphasizing a deep understanding of each stage of the RAG pipeline (data preprocessing, chunking, embedding, indexing, retrieval, generation, synthetic QA generation with k-means).
- **Technical Stack:**
  - Embedding: `intfloat/multilingual-e5-large-instruct`
  - Vector Store: FAISS (`IndexFlatIP`)
  - LLM: `meta-llama/Llama-3.3-70B-Instruct-Turbo-Free` (via Together AI)
  - Custom Python scripts for all pipeline stages.
- **Functionality:** End-to-end RAG for UCY web data; robust synthetic QA dataset generation and evaluation.
- **Strengths:**
  - **Deep Understanding:** Provided unparalleled insight into core RAG mechanics.
  - **Customizability:** Full control over every component, allowing for fine-tuning and

experimentation.
    - **Innovative QA Generation:** The k-means clustering approach for diverse chunk selection for QA generation was a notable strength.
- **Weaknesses:**
    - **Development Effort:** Building from scratch is time-consuming and requires significant engineering effort.
    - **Scalability & Maintenance:** Custom solutions can be harder to scale and maintain compared to systems built on established frameworks.
    - **Limited Abstraction:** Lacks the higher-level abstractions for complex agentic behaviors or tool integration found in frameworks.

6.2. Langchain/LangGraph Agent Implementation

- **Methodology:** Leveraged Langchain for RAG components and LangGraph for creating a stateful, multi-tool agent. Focused on modularity and routing queries to specialized tools.
- **Technical Stack:**
    - Embedding: `VoyageAI voyage-3-lite`
    - Vector Store: Pinecone (multiple indexes for different data types)
    - LLM: OpenAI `gpt-4o`
    - Langchain for chains, retrievers, prompts; LangGraph for agent orchestration.
- **Functionality:** Agentic system with specialized nodes (General Question, Schedule Question, Form Node) handling diverse queries about UCY. Maintained conversation history.
- **Strengths:**
    - **Rapid Development:** Langchain's pre-built components significantly sped up development.
    - **Rich Ecosystem:** Access to a wide array of Langchain integrations and tools.
    - **Agentic Capabilities:** LangGraph enabled sophisticated agentic behavior, including tool selection and state management.
    - **Modularity:** Tool-based architecture allows for easy extension with new functionalities or data sources.
- **Weaknesses:**
    - **Framework Dependence:** Reliant on Langchain/LangGraph's abstractions, which can sometimes be complex or opaque.
    - **Debugging:** Debugging complex LangGraph flows can be challenging.
    - **Cost:** Use of proprietary models (VoyageAI, OpenAI GPT-4o) and services (Pinecone) incurs costs.

6.3. OpenAI Agents SDK Agent Implementation

- **Methodology:** Utilized the OpenAI Agents Framework for a declarative and modular approach to orchestrating sub-agents and managing state. Emphasized advanced techniques like LLM-based document grading.
- **Technical Stack:**
    - Embedding: Ollama `nomic-embed-text` (local)
    - Vector Store: FAISS (index built via `warmup.ipynb`)
    - LLM (for grading/orchestration): OpenAI `gpt-3.5-turbo` or `gpt-4.1-mini`
    - OpenAI Agents SDK for agent definition, orchestration, and tool management.

- **Functionality:** Modular RAG system with a Triage Agent orchestrating sub-agents/tools (e.g., retrieval, document grading). Managed short-term memory with `RagAgentContext`.
- **Strengths:**
  - **Declarative Structure:** Simplifies the definition and coordination of complex agent workflows.
  - **Advanced Orchestration:** Native support for tool-calling, stateful context, and agent handoff.
  - **Modularity for Complex Systems:** Well-suited for building intricate multi-agent systems.
  - **Innovative Techniques:** Implemented LLM-based document grading for relevance and XML tags for prompt engineering.
  - **Local Embeddings:** Use of Ollama allows for local, cost-free embedding generation.
- **Weaknesses:**
  - **Framework Specificity:** Tied to the OpenAI Agents SDK, which might be less mature or have a smaller community than Langchain.
  - **Setup Complexity:** Requires setup of Ollama for local embeddings and potentially a more involved understanding of the SDK's specific concepts.

6.4. Comparison of High-Level Frameworks (Langchain vs. OpenAI Agents SDK) Based on the team's implementations:

- **Langchain:**
  - **Strengths:** Mature, extensive documentation, large community, vast number of integrations, flexible (can build both simple chains and complex agents). LangGraph adds powerful stateful agent capabilities.
  - **Weaknesses:** Can have a steeper learning curve for complex agentic patterns due to its "many ways to do one thing" philosophy. Debugging intricate chains/graphs can be challenging.
- **OpenAI Agents SDK:**
  - **Strengths:** More recent, specifically designed for building agents with a more declarative and potentially cleaner structure for orchestration. Good integration with OpenAI models and tools. Pydantic-based state management is robust.
  - **Weaknesses:** Newer framework, potentially less community support and fewer third-party integrations compared to Langchain. Might be more opinionated in its architectural approach.

6.5. Overall Reflections A direct quantitative performance comparison is difficult due to differing datasets, evaluation metrics, and the specific focus of each implementation. The low-level work excelled in deep mechanical understanding and rigorous self-evaluation. The Langchain/LangGraph exploration demonstrated rapid, modular agent development with existing data integration. The OpenAI Agents SDK exploration showcased advanced orchestration and cutting-edge RAG refinement techniques.

The development experience varied: The low-level implementation was intensive but highly educational. The Langchain/LangGraph work benefited from Langchain's abstractions for quicker setup of RAG components, with complexity arising in LangGraph orchestration. The OpenAI Agents SDK work leveraged its structure, potentially offering a more streamlined path for specific agent designs but requiring familiarity with its paradigm. Flexibility seemed high in the custom build, while Langchain offers broad flexibility through its vast components. OpenAI Agents SDK offers flexibility within its agent-centric model. Scalability often favors framework-based approaches over purely custom ones, though specific framework choices also impact this.

Each approach successfully contributed to the project's overall goal of understanding and implementing RAG technology for the UCY AI Assistant, providing a rich set of learnings and potential components for

future development.

7. Next Steps for the Team

The explorations undertaken by the team have provided a wealth of knowledge and practical experience. To build upon this foundation and advance the UCY AI Assistant project, the following actionable next steps are proposed:

1. **Develop a Common Evaluation Framework and Dataset:**
   - **Challenge:** A key difficulty in comparing the different approaches was the lack of a unified evaluation standard.
   - **Action:** Create a standardized, representative dataset of UCY-related questions and corresponding ground-truth answers/documents. This dataset should cover diverse topics and question types relevant to UCY students and staff.
   - **Metrics:** Define a common set of evaluation metrics for both retrieval (e.g., MRR, Precision@K) and generation (e.g., RAGAs metrics like faithfulness, answer relevancy, context precision/recall, or human evaluation). This will allow for more direct and meaningful comparisons of future iterations or hybrid approaches.
2. **Synthesize Learnings and Integrate Components:**
   - **Low-Level Work:** The insights on data preprocessing, the impact of question quality, and the k-means approach for diverse QA dataset generation are valuable. The low-level understanding of bottlenecks can inform optimization.
   - **Langchain/LangGraph Work:** The modular tool-based architecture using LangGraph is powerful for handling diverse query types. The experience with Pinecone and VoyageAI embeddings provides a benchmark for production-grade vector DBs and embedding models. The data processing and description generation scripts, and the schedule data handling, are valuable assets.
   - **OpenAI Agents SDK Work:** The use of OpenAI Agents SDK for orchestration, LLM-based document grading for relevance, XML-tagged prompting, and structured output with Pydantic are advanced techniques that can significantly improve precision and control. The use of local Ollama embeddings is also a key consideration.
   - **Action:** Hold workshops to share detailed findings and code. Identify specific components or techniques from each approach that can be merged or adapted. For example, the document grading from the OpenAI Agents SDK work could be integrated as a tool in the LangGraph agent, or the data preprocessing pipeline from the low-level work could feed into any of the higher-level systems.
3. **Decide on a Primary Framework/Architecture for Further Development:**
   - Based on the common evaluation framework, team expertise, development speed, scalability requirements, and desired features (e.g., complexity of agentic behavior, state management needs), make an informed decision on whether to primarily pursue Langchain/LangGraph, OpenAI Agents SDK, or potentially a hybrid approach for the core UCY AI Assistant.
   - This decision should consider long-term maintainability and the UCY's technical ecosystem.
4. **Enhancements to Data Quality and Coverage:**
   - Continuously update and expand the knowledge base with the latest UCY information (new courses, policies, events, website changes).
   - Refine data preprocessing pipelines (building on Chrysis's and Maria's work) to ensure high-quality, clean data.
   - Explore automated methods for detecting and ingesting new or updated information.

5. **Advanced Retrieval and Generation Strategies:**
    - **Retrieval:** Experiment with hybrid search (sparse + dense retrieval), re-ranking models, and query transformation techniques (e.g., HyDE, query rewriting) to improve retrieval accuracy, especially for complex or ambiguous queries.
    - **Generation:** Explore different LLMs for generation, focusing on models that excel at synthesis, factuality, and adhering to instructions. Refine prompting strategies further, potentially incorporating dynamic few-shot examples.
    - **Context Management:** Investigate more sophisticated methods for context window management and condensing information from multiple retrieved chunks.
6. **User Interface and Feedback Loop:**
    - Develop a user-friendly interface (e.g., a web application) for the UCY AI Assistant.
    - Implement a feedback mechanism where users can rate the quality and helpfulness of answers. This feedback is invaluable for iterative improvement and identifying areas needing attention.

By systematically addressing these next steps, the team can leverage the diverse explorations into a cohesive and increasingly capable UCY AI Assistant, ultimately fulfilling the project's objective of improving information access at the University of Cyprus.

8. Conclusion

The development of the UCY AI Assistant has been a comprehensive journey, systematically progressing from foundational, low-level RAG implementation to the exploration of sophisticated, production-grade frameworks. The initial low-level work laid a crucial groundwork, providing deep insights into the core mechanics of RAG systems and the importance of meticulous data handling and evaluation. This foundational understanding enabled the team to effectively explore higher-level solutions.

The Langchain/LangGraph exploration demonstrated the power of modular, tool-based agentic architectures for handling diverse information needs, supported by collaborative efforts in data preparation. The investigation into the OpenAI Agents SDK highlighted advanced orchestration capabilities, state management, and innovative techniques like LLM-based document grading, showcasing another robust path for developing complex RAG agents.

The team's multi-faceted approach, while presenting challenges for direct quantitative comparison, yielded a rich tapestry of learnings. Key takeaways include the critical impact of data quality and preprocessing, the nuanced benefits of different embedding and LLM choices, the value of synthetic dataset generation for evaluation, and the distinct advantages offered by various frameworks in terms of development speed, flexibility, and agentic sophistication.

The collective efforts of the team have significantly advanced the UCY AI Assistant project. While each exploration pursued distinct methodologies, their contributions collectively paint a clear picture of the possibilities and complexities of RAG technology. The project has not only built functional prototypes but also fostered a deep, practical understanding within the team. The insights gained from exploring these different paths provide a strong basis for making informed decisions about the future architecture and refinement of an AI assistant that holds great potential to enhance the daily lives of students and staff at the University of Cyprus by making information more accessible and actionable.