**CISC 691: A06: Building the AI Agent of Your Choice!**

1. **Agent Purpose, Use Case, and Problem Solved**

The project develops a sophisticated virtual agent which aids e-commerce consumers to quickly obtain trustworthy insights from Amazon product reviews. This system tackles the issue of information overload users experience when they try to evaluate products based on a plethora of unstructured reviews. The agent combines customer feedback with structured data like ratings and prices to provide users with clear summaries that help them make informed purchase choices. Interactive dialogue features in the solution show confidence scores with follow-up questions that enhance trust and maintain user engagement throughout multi-turn conversations. The agent proves to be a valuable resource for both shoppers, customer service personnel and personal assistants operating in e-commerce fields.

1. **Design Pattern, Architecture, Workflow and Framework Selection**

The Retrieval-Augmented Generation (RAG) design pattern forms the structure of our system because it integrates a mechanism for document retrieval with a generative model. We use ChromaDB as a vector database to store and access embeddings from preprocessed Amazon reviews. The system uses OpenAI’s GPT-4 to create a final answer from the retrieved data which contains review text along with structured product data.

The overall architecture is hybrid and layered. The raw Amazon CSV first undergoes ingestion before moving through preprocessing and division into smaller manageable segments. Transformer-based models from libraries like Hugging Face Transformers transform chunks into vector embeddings which are stored in ChromaDB. The retrieval engine processes incoming FastAPI endpoint queries by fetching top relevant documents and calculating a composite confidence score. The system combines retrieved text with metadata to create a prompt for GPT-4 which generates the final answer. The system architecture features strong error management through asynchronous tasks alongside retry features and customizable settings to handle different levels of demand and unpredictable real-world conditions.

*Workflow Design:*

A diagram of a process

AI-generated content may be incorrect.

*SVG Link for Workflow:* [*https://www.mermaidchart.com/raw/cd0dc186-4222-40dd-a729-0c28a8bb35e9?theme=light&version=v0.1&format=svg*](https://www.mermaidchart.com/raw/cd0dc186-4222-40dd-a729-0c28a8bb35e9?theme=light&version=v0.1&format=svg)

Below is an excerpt of the core code used to initialize the configuration, load the OpenAI API key, and define the FastAPI application along with request/response models:

***Code Snippet:***

A screen shot of a computer

AI-generated content may be incorrect.

This segment of code initiates the foundational setup for the agent's operations. The ConfigManager manages configuration details while FastAPI creates the web server and its endpoints. By implementing asynchronous libraries such as asyncio with proper logging practices, we achieve both system responsiveness and debuggability during high-load situations.

1. **Workflow Design**

The agent’s workflow operates through a sequence of modular steps which follow one after another. The agent’s workflow starts by consuming raw data from an Amazon CSV file then processes it. The text undergoes tokenization and may be divided into parts when needed to stay within model size restrictions. The text chunks undergo an embedding process, and the results are saved in ChromaDB which supports rapid similarity searches.

During query processing the retrieval function locates and retrieves the top-k documents with highest relevance. The provided code snippet demonstrates the updated retrieval function which now utilizes pre-existing RAG pipeline functions instead of dummy data.

A computer screen shot of text

AI-generated content may be incorrect.

This function creates a ChromaDBRetriever instance with the provided configuration parameters before querying the vector database to fetch relevant document pieces. The design uses asynchronous operations to reduce blocking time during I/O process waits.

The system computes a composite confidence score from the similar scores of the documents it retrieves. This function determines the score by taking the average of document scores and transforming them into percentages without including penalty-based variance in the below code.A screen shot of a computer

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

We combine the structured data from the retrieved documents into one context string before sending it to GPT-4. Each document must contain product\_name field as well as review information rating and price to be processed correctly within this function. Missing fields trigger the error handler during processing:A computer code on a black background

AI-generated content may be incorrect.

For generating the final answer, an asynchronous function calls GPT-4 with a retry mechanism incorporating exponential backoff. This ensures reliability when external API calls experience intermittent failures:

A computer screen shot of text

AI-generated content may be incorrect.

Additionally, the system generates follow-up questions to maintain ongoing dialogue and uses a helper function to format the answer for improved readability:

A screen shot of a computer program

AI-generated content may be incorrect.

Finally, the FastAPI endpoint brings together all these functions in a robust, error-handled workflow. When a query is received, the endpoint retrieves documents, calculates the confidence score, integrates data, generates the response via GPT-4, formats the answer, and generates follow-up questions:

A screen shot of a computer program

AI-generated content may be incorrect.

The entire system is bootstrapped by a main function that runs the server using Uvicorn with auto-reload enabled during development:

A black screen with white text

AI-generated content may be incorrect.

This code demonstrates how each component of the agent’s workflow is implemented, from configuration and asynchronous processing to error handling and response formatting.

**Postman App API:**

In the next part we explore how the POSTMAN app helps to retrieve the answers we need using the AI Agent built.

**Server Setup**

Before testing the API in Postman, you must ensure that your FastAPI server is running. This is achieved by starting the server with Uvicorn using a command like uvicorn agent:app --host 127.0.0.1 --port 8000 --reload in your terminal. This command launches the server on your local machine at 127.0.0.1:8000, which prepares the backend to accept incoming requests.

**Creating a New Request in Postman**

After starting the server, open Postman to create a new API request. In Postman, select the option to create a new request, and set the HTTP method to **POST**. You then provide the URL http://127.0.0.1:8000/query in the address field, ensuring that the request is routed to the correct endpoint that processes queries on your API.

**Setting Request Headers**

Within Postman, navigate to the "Headers" tab and add a header with the key Content-Type and the value application/Json. This header is crucial because it informs the server that the payload being sent is formatted in JSON, ensuring the FastAPI server can correctly parse the incoming data.

**Defining the Request Body**

Switch to the "Body" tab in Postman and select the "raw" option, making sure to set the format to JSON from the dropdown menu. Here, you enter the JSON payload according to the QueryRequest model defined in the code. For example, a valid payload might look like this:

A screenshot of a computer

AI-generated content may be incorrect.

This JSON structure supplies the necessary data, where the "query" field is mandatory, and "session\_id" is optional for maintaining context in multi-turn conversations.

**Sending the Request**

After configuring the URL headers and request body correctly, you need to click the Send button within Postman. The action forwards your request to the active FastAPI server. The server first confirms that the JSON payload is valid before starting its internal operations which include document retrieval and confidence score calculation alongside structured data integration to produce an answer.

**Processing the Request on the Server**

The server API processes incoming queries by asynchronously fetching relevant documents from a vector database through the ChromaDBRetriever class. The system calculates a composite confidence score from document scores and merges structured data like product details into a single context string. The system merges the original query with the created context string before sending it to the OpenAI API with the GPT-4 model to generate a detailed response. During operation the server maintains records of essential processing steps and addresses any errors that arise smoothly.

**Reviewing the Response**

The API produces a JSON response following the QueryResponse model after processing. When viewing results in Postman you find a JSON response that holds the generated answer and includes the confidence score of document relevance along with follow-up questions. The review process demonstrates that the system executed all tasks properly from document collection through to final response creation. Postman shows the correct HTTP error codes defined by the API when errors happen.

**Response Input & Output:**

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

1. **Challenges and Their Resolutions**

A major development problem involved combining unstructured review text with structured product information. Since product names were absent from the initial dummy data the ingestion process required enhancement to pull this information from the CSV file. To solve the problem we updated ingestion and integration functions, so they used a new key named "product\_name" and implemented try-except blocks to manage missing keys without errors. We needed to build resilience into our external API calls to GPT-4. The generate\_gpt4\_response function incorporated a retry loop with exponential backoff to address this issue. The code extensively implemented asynchronous operations for I/O processes to prevent blocking and enhance system responsiveness. The agent became more robust and reliable because of these enhancements together with modular code organization and comprehensive logging.

1. **Reflection and Areas for Future Improvement**

The system functions effectively to gather product reviews from Amazon data while transforming them into meaningful insights. However, further improvements are possible. Future retrieval accuracy and confidence levels will benefit from the implementation of advanced domain-specific embedding models. The use of advanced re-ranking through cross-encoders enhances answer precision by providing a richer context. The dialogue management system could develop capabilities for complex multi-turn conversations through the application of reinforcement learning which utilizes real-time user feedback for response refinement. By implementing performance optimizations such as data caching and batch processing for multiple queries we can decrease latency and enhance user experience during production.

**Conclusion:**

The report analysis presents an advanced RAG-enabled virtual assistant created to process e-commerce evaluations. The modular FastAPI framework combines advanced retrieval techniques and structured data processing with asynchronous operations and robust error handling to deliver real-time insights with rich context from Amazon reviews. The solution resolves the problem of information overload while creating groundwork for future improvements in embedding precision as well as dialogue management and system performance optimization. Through ongoing development efforts the agent will deliver progressively more precise and engaging personalized experiences to users.

**GitHub Repository Link:** [**https://github.com/VVPPower/CISC-691\_A06\_AI-Agent**](https://github.com/VVPPower/CISC-691_A06_AI-Agent)

**References:**

* 1. Professor. Don Ohara assignment of RAG pipeline is used for building AI Agent.
  2. Professor. Majid Shaalan is input from assignment 05.
  3. ChatGPT API.
  4. <https://www.postman.com/>
  5. <https://www.mermaidchart.com/>
  6. Jeong, C. (2024). A Graph-Agent-Based Approach to Enhancing Knowledge-Based QA with Advanced RAG. *Knowledge Management Research*, 99-119.