



Syracuse University

Project Report

Advanced Chatbot Solution Using Azure

IST 615- Cloud Management

School of Information Studies

Team Members (Group-1)

- Amisha Gangwar
- Satwik Mrityunjay Belaldavar
- Siddharth Bose
- Varshin Hariharan Bhaskaran

Project Goals & Objectives

The primary goal is to develop an advanced chatbot capable of understanding and processing natural language queries, providing intelligent responses, and integrating with external systems for enhanced user engagement. Our chatbot aims to serve as a first point of contact for customer support, significantly reducing response times and improving customer satisfaction.

Objectives:

Natural Language Processing (NLP) Integration: Integrate and fine-tune a state-of-the-art NLP model to accurately interpret financial jargon and user queries.

Document Parsing and Summarization: Develop a robust mechanism to parse various formats of financial documents and utilize AI to extract and condense the key information into accurate summaries.

Cloud Infrastructure Scalability: Create a scalable cloud infrastructure capable of handling a high volume of concurrent users and large documents without performance degradation.

Knowledge Base Development: Construct a dynamic knowledge base that the chatbot can reference, which is continuously updated with the latest financial data and insights.

Cloud Services and Tools

1. User Interface (Streamlit): An open-source app framework for Machine Learning and Data Science teams. It was used to create a user-friendly web interface for users to interact with the chatbot.

2. Azure Virtual Machines (VM): Scalable cloud computing services from Microsoft Azure that provide on-demand, scalable computing resources. In this project, they hosted the Streamlit application and handled processing and summarization of documents.

3. Azure Bot Services: A managed platform for developing AI chatbots. It orchestrates the conversational flow of the chatbot and integrates with other cognitive services to provide intelligent responses.

4. Azure AI Search: A cloud search service with built-in AI capabilities that enrich all types of information to easily identify and explore relevant content at scale. It was used to index and search financial document contents.

5. Azure Cosmos DB: A globally distributed, multi-model database service that offers scalability and low-latency access to data. It was used for storing structured transactional data and logs.

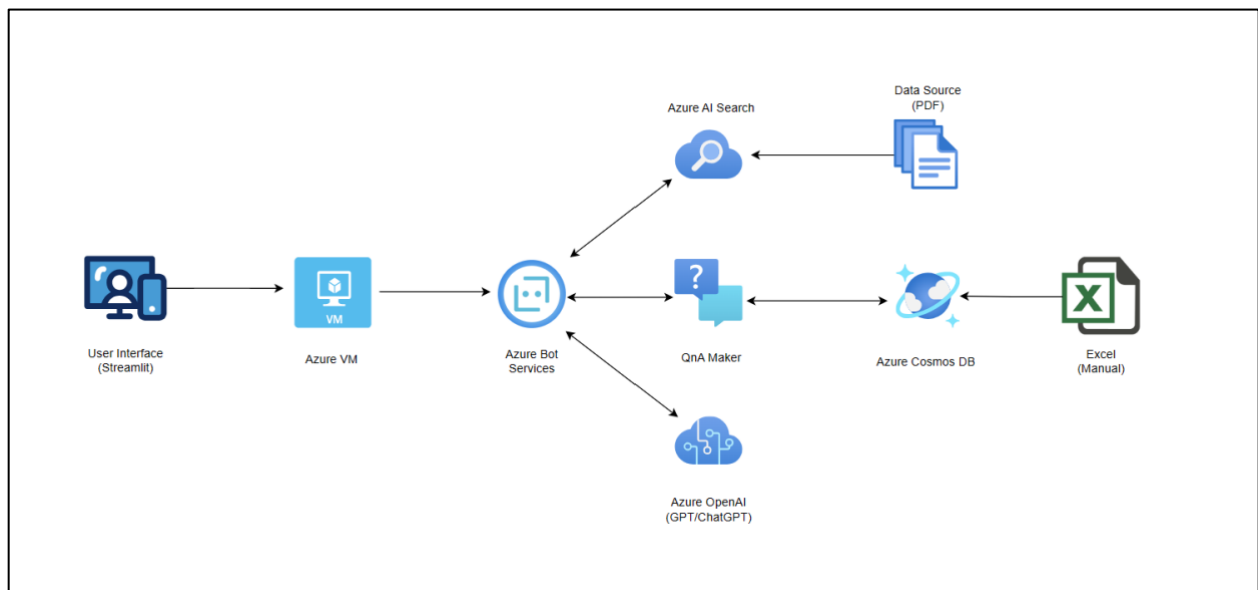
6. Azure QnA Maker: A cloud-based API service by Microsoft Azure that creates a conversational layer over your data. It was utilized to create a responsive knowledge base for the chatbot.

7. Azure OpenAI: Cloud-based AI services offering access to OpenAI's language models, like GPT-3 or ChatGPT. It was leveraged for advanced natural language understanding and text generation to create summaries from financial documents.

Additional Resources Needed

- **Development Tools:** Visual Studio Code, Bot Framework Emulator for local testing.
- **Database Services:** Azure Cosmos DB for storing user conversations and preferences.
- **Monitoring and Analytics:** Azure Application Insights for performance monitoring and user interaction analytics.
- **Retrieval-Augmented Generation (RAG) AI:** The integration of RAG AI requires access to advanced machine learning models and computational resources. Azure Machine Learning and Azure Cognitive Services will be pivotal in implementing RAG AI, enabling our chatbot to leverage vast amounts of information and generate contextually relevant responses in real-time. This will enhance the chatbot's ability to understand and respond to complex queries more effectively.

Architecture Diagram



Solution Overview

Our sophisticated chatbot harnesses Azure Bot Service to facilitate seamless user interactions and deploy conversational interfaces. It utilizes the natural language processing prowess of Azure OpenAI, including GPT/ChatGPT models, to analyze and summarize complex financial documents. The Azure AI Search is instrumental in swiftly retrieving data from the uploaded documents, and the integration with Azure Blob Storage ensures secure document management. For prompt responses to standard inquiries, Azure QnA Maker provides the chatbot with a comprehensive knowledge base. All interactions and transactional data are meticulously logged in Azure Cosmos DB, which supports the chatbot's adaptive learning capabilities. Monitoring through Azure Application Insights enables detailed analysis of the chatbot's performance, guiding ongoing enhancements. This results in a continually evolving, user-centric chatbot that simplifies financial document summarization, optimizing data handling, and aiding decision-making processes.

Workflow:

1. **Document Upload:** Users uploaded financial PDF documents via the Streamlit interface, which were sent to the Azure VM.
2. **Processing:** The Azure VM conducted initial document processing and then stored the documents in Blob Storage.
3. **Indexing:** Azure AI Search indexed the documents, making the data searchable for query processing.
4. **Query Handling:** Users entered queries to the chatbot via the Streamlit interface, which were managed by Azure Bot Services.
5. **Information Retrieval:** The chatbot utilized QnA Maker for quick answers or Azure OpenAI for complex queries and summary generation.

6. **Data Storage:** Cosmos DB recorded all transactional data for future analytics and chatbot training purposes.
7. **User Interaction:** The chatbot communicated responses back to the user through the Streamlit interface, completing the interaction cycle.

Configuration of cloud services and resources

1. User Interface (Streamlit):

- **Designed Interface:** A responsive web interface was created with Streamlit, which allowed users to upload financial documents and interact with the chatbot.
- **User Interaction:** Input forms for text queries and file upload capabilities for document analysis were implemented.
- **Response Display:** Sections were set up to display chatbot responses, including summaries and any other relevant data.

2. Azure Virtual Machines (VM):

- **VM Setup:** An appropriately sized VM was chosen based on the expected computational needs for processing and summarizing documents.
- **Software Configuration:** Necessary software, like Python and required libraries for Streamlit, was installed, and the environment was configured for integration with Azure services.
- **Endpoint Creation:** HTTP endpoints on the VM were set up for receiving requests from the Streamlit UI and sending responses back.

3. Azure Bot Services:

- **Bot Registration:** A new bot was registered on the Azure portal and necessary credentials were obtained.
- **Bot Framework:** The Azure Bot Framework SDK was used to build the chatbot logic, handle session states, and manage conversation flow.

- **Integrations:** Connectors were set up to interface with QnA Maker and Azure OpenAI for processing queries.

4. Azure AI Search:

- **Data Ingestion:** The indexing pipeline was configured to ingest documents from Azure Blob Storage where PDFs were stored.
- **Index Configuration:** The index schema was defined with fields that represented the document data to be searchable.
- **Query Setup:** Queries were created to retrieve information from the index based on user inputs from the chatbot.

5. Data Source (PDF):

- **Storage Account:** An Azure Blob Storage account and container were created for storing uploaded PDFs.
- **Security:** Security measures, like SAS tokens, were implemented to ensure safe document upload and access.
- **Connectivity:** The storage was made accessible by Azure AI Search for indexing and by the VM for processing.

6. Azure Cosmos DB:

- **Database Setup:** A new Cosmos DB account was created, and a database and container structure was defined that aligned with the data intended to be stored.
- **Data Model:** A data model was designed for storing structured information extracted from financial documents.
- **API Selection:** The appropriate API for Cosmos DB was chosen; SQL API was identified as the most straightforward for this use case.

7. Excel (Manual):

- **Data Organization:** Excel sheets were structured to include summaries and key data points that could be ingested into Cosmos DB or used for setting up our basic bot.

- **Upload Process:** A process was established for regularly importing this data into Cosmos DB or an indexing pipeline for searchability.

8. Azure QnA Maker:

- **Knowledge Base Setup:** A new QnA Maker service was created and a knowledge base was constructed using FAQs, financial data, and manual summaries.

- **Integration:** The QnA Maker related to the Azure Bot Services to provide quick, canned responses for common queries.

9. Azure OpenAI:

- **Service Provisioning:** An Azure OpenAI service account was set up and access to GPT models was configured.

- **Model Training:** The model was trained using financial data and documents to understand and generate industry-specific summaries.

- **API Integration:** API calls were developed within the bot framework to send queries to and receive responses from the GPT model.

Tasks Completed:

Here is a list of tasks that were completed for the AI chatbot project:

1. User Interface Development with Streamlit: A responsive web application was created using Streamlit for users to upload financial documents and interact with the AI chatbot.

2. Virtual Machine Provisioning: An Azure VM was set up to host the application and provide the computational power required for processing and summarizing the financial documents.

3. Chatbot Registration and Configuration: A chatbot was registered and configured using Azure Bot Services, establishing the conversational framework and integrating cognitive services for intelligent responses.

4. Document Indexing System Setup: Azure AI Search was configured to index financial documents uploaded by users, enabling efficient data retrieval for the chatbot.

5. Blob Storage for Document Management: Azure Blob Storage was established to securely store uploaded financial documents, providing a scalable solution for document management.

6. Cosmos DB Integration for Data Storage: Azure Cosmos DB was integrated to store structured data, including transaction logs, aiding in the bot's learning and information retrieval.

7. Excel Data Integration for bot setup: Summaries and key data points were organized in Excel spreadsheets for ingestion into the system, assisting in the knowledge base development.

8. Knowledge Base Creation with QnA Maker: A knowledge base was built using QnA Maker, allowing the chatbot to quickly provide answers to frequently asked questions.

9. Natural Language Processing with Azure OpenAI: Azure OpenAI services were employed to process user queries and generate accurate summaries from the financial documents using advanced NLP techniques.

10. User Query and Document Processing Workflow: A workflow was established that started with users uploading documents, the system processing these documents, and the chatbot presenting summarized information.

11. Monitoring and Logging Setup: Azure Application Insights was set up for monitoring and logging the chatbot's interactions, performance, and user engagement.

12. Continuous Improvement Mechanism: Feedback loops were implemented to capture user feedback and continuously refine the chatbot's performance based on this input.

13. Security Protocols Implementation: Data security measures were put in place, including the configuration of firewalls, IAM, and encryption practices to protect sensitive financial information.

14. Compliance Checks and Auditing: Procedures and tools were established to ensure the project complied with relevant data protection and privacy regulations.

15. Performance Optimization: Ongoing tasks included monitoring system performance and tuning resources to handle the expected load and to optimize the user experience.

Issues encountered and lessons learned

During the initial stages of our project, we aimed to develop a chatbot that could extract information directly from the Microsoft Docs Azure repository([MicrosoftDocs GitHub repository](#)) using a data crawler. However, we faced several challenges, including data quality, scalability, and complex integration issues with existing IT infrastructures like Microsoft Teams. These difficulties highlighted administrative and access-level constraints within Teams, as well as complications in real-time data synchronization and user authentication. As a result, we pivoted to a chatbot focused on summarizing financial documents and chose Streamlit for our frontend development. Streamlit provided a more straightforward and flexible user interface, allowing us to overcome the integration hurdles with Microsoft Teams and focus on delivering a user-friendly experience free from platform-specific restrictions. This strategic shift not only simplified deployment but also enhanced our chatbot's functionality and scalability.

Issues Encountered:

Crawler Efficiency and Scalability: Our initial data crawler struggled with efficiently handling large volumes of data. Scalability became an issue as the crawler was unable to process data at the speed and accuracy required.

Data Quality and Accessibility: The quality of data retrieved by the crawler varied significantly, and access to consistent and reliable data sources was more problematic than anticipated. This inconsistency affected the reliability of the information the chatbot provided.

Integration Challenges: Integrating the crawler with existing IT infrastructures proved complex, particularly concerning data consistency and real-time processing needs. This led to difficulties in ensuring that the chatbot could provide timely and accurate responses based on the latest available data.

Administrative and Access-Level Constraints: We encountered complex administrative barriers and access-level restrictions within Microsoft Teams that hindered the seamless integration of our chatbot. Navigating these constraints proved to be more challenging than anticipated.

Complex API and Permissions: The Microsoft Teams API presented complexities in terms of required permissions and the handling of data, which affected our ability to deploy features effectively and efficiently.

User Authentication Issues: Integrating user authentication mechanisms between Microsoft Teams and our backend systems was problematic, leading to delays and user access issues.

Real-Time Data Synchronization: Achieving real-time data synchronization between the chatbot's responses and Teams' dynamic environment was difficult, impacting the user experience negatively.

Lessons Learned

Need for Robust Data Management: This challenge underscored the importance of having robust data management systems that ensure data quality, consistency, and compliance with regulatory standards.

Focus on Core Competencies: The difficulties highlighted the need to focus on core competencies. By pivoting to a financial document summarization solution, we could leverage our strengths in natural language processing and AI more effectively.

Importance of Scalable Solutions: The scalability issues taught us the importance of designing solutions that are not only effective but also scalable from the outset, particularly in handling large datasets or integrating with various platforms.

Enhanced Integration Strategies: We learned the importance of developing more sophisticated integration strategies, especially when dealing with complex IT infrastructures like those found in large organizations.

User-Centric Approach: Ensuring the technology aligns with user needs and existing workflows is essential. Our initial approach overlooked some of the practical challenges of deploying new technologies within established systems.

Simplification of Deployment: These challenges taught us the importance of simplifying our deployment strategy. Streamlit offered a more straightforward and flexible way to develop and deploy our chatbot's user interface without the complexities of integrating with a robust platform like Microsoft Teams.

Enhanced Control Over UI: Streamlit provided us with greater control over the user interface design and interaction flow, which was crucial for delivering a user-friendly experience that could still meet our technical requirements.

Rapid Prototyping and Iteration: Using Streamlit allowed for faster prototyping and iterative testing, which was essential for a rapid development cycle and immediate feedback incorporation.

Independence from Platform-Specific Restrictions: By moving away from Microsoft Teams, we avoided the platform-specific restrictions and could implement features that were previously limited by Teams' capabilities and policies.

Focused User Experience: Streamlit enabled us to focus on the core functionalities of our chatbot without being tied down by the infrastructural and administrative constraints of an enterprise platform like Microsoft Teams.

Timeline:

March 25, 2024: Project Kick-Off

- Initial team meeting and project planning are conducted to outline objectives and assign responsibilities. Emphasis is placed on flexibility and adaptability in our approach, given the complexities anticipated with new technology integrations.

April 1, 2024: Basic Chatbot Development Completed

- Completion of the foundational development of the chatbot using Azure Bot Services and LUIS. This phase includes initial testing to identify any major integration issues with the core Azure services.

April 8, 2024: Basic Knowledge Base Enhancement

- The team enhances the foundational knowledge base to improve the chatbot's accuracy in handling FAQs, incorporating lessons learned from early tests about data quality and handling.

April 12, 2024: Pivot Strategy Review and RAG AI Integration Planning

- Given the challenges encountered with Microsoft Teams integration and data crawler issues, the team reviews the project strategy. Planning for the integration of Retrieval-Augmented Generation

AI begins, aiming to enrich the chatbot's response capabilities with a focus on scalability and enhanced natural language processing.

April 15, 2024: Streamlit Frontend Development Kick-Off

- Based on the decision to pivot from Microsoft Teams to Streamlit for frontend development, this phase focuses on setting up a user-friendly interface for uploading documents and interacting with the chatbot.

April 19, 2024: RAG AI Integration Execution

- The team starts executing the planned integration of RAG AI to enhance the chatbot's functionality, taking into account the need for a robust integration strategy that accommodates complex IT infrastructures.

April 22, 2024: Progress Review Meeting

- A follow-up meeting to discuss the ongoing RAG AI integration process, evaluate the progress, and make necessary course corrections to ensure alignment with project goals. This includes addressing any issues from the Streamlit integration and data handling adjustments.

April 24, 2024: Integration and Testing Phase

- Comprehensive testing of the RAG AI integration and the new Streamlit frontend to validate the chatbot's enhanced capabilities and performance across various scenarios. This testing is critical to ensure that all components work seamlessly together.

April 26, 2024: Final Review & Adjustments

- The team conducts a final review of the chatbot, making any last-minute adjustments based on testing feedback and preparing for the project presentation. Special attention is given to user experience and interface simplicity.

April 29, 2024: Project Presentation

- The project is presented, showcasing the advanced functionalities of the chatbot and the impact of incorporating RAG AI, Streamlit, and other Azure services. This presentation highlights the

journey from the initial challenges to the successful pivot and integration of advanced AI technologies.

Conclusion

The journey of developing our advanced chatbot solution using Azure technologies has been both challenging and enlightening, pushing the boundaries of what we thought was feasible and teaching us valuable lessons along the way. Our project aimed to create a state-of-the-art AI chatbot—a tool that would not only enhance user engagement but also streamline customer support processes.

Initially, we attempted to develop a chatbot that could directly extract information using a data crawler from the Microsoft Docs Azure repository. However, we quickly encountered significant challenges related to data quality, scalability, and integration with Microsoft Teams. These challenges were compounded by administrative and access-level constraints, leading to a pivotal decision to pivot our approach.

We shifted our focus to developing a chatbot that specializes in summarizing financial documents, which allowed us to utilize our strengths in natural language processing more effectively. The integration complexities with Microsoft Teams prompted us to adopt Streamlit for our front-end development, providing us with a simplified and more flexible user interface. This change not only facilitated easier deployment but also enhanced our chatbot's functionality and user experience.


The core of our project utilized several Azure services: Azure Bot Services orchestrated the conversational flow; Azure AI Search indexed the document contents; Azure Cosmos DB managed transactional data; and Azure OpenAI powered the natural language understanding capabilities of the chatbot. Each component was integral to creating a seamless and effective user experience.

From this project, we learned the importance of flexibility and adaptability in technology projects—especially when dealing with emerging technologies and complex IT environments. The ability to pivot and leverage different technologies like Streamlit and Azure OpenAI proved crucial in overcoming the initial hurdles. We also recognized the need for a robust testing and feedback loop, ensuring that our chatbot continuously evolves and improves based on user interaction.

As we look to the future, our team is excited to continue refining our chatbot. We plan to integrate more advanced AI capabilities, such as Retrieval-Augmented Generation (RAG) AI, to further enhance the chatbot's ability to understand and respond to complex queries. The goal is to not just meet but exceed user expectations, making our chatbot a benchmark for innovation in customer support technology.

This project has not only been a testament to our team's resilience and capability to navigate through challenges but also a clear indicator of the transformative potential of AI and cloud technologies in reshaping how businesses interact with their customers. We remain committed to pushing the envelope, ensuring that our solutions are at the forefront of technological advancement and customer satisfaction.

APPENDIX:

 **Finance GPT**

Explore financial documents with the power of GPT.


Input your prompt here

What are the main financial highlights from the latest annual report?

Enter a prompt related to finance to get insights from your document collection.

The main financial highlights from the latest annual report include a 22% increase in total operating expenses, a 76% increase in income tax expense, and a 56% increase in NPAT (Net Profit After Tax). Additionally, there was a 32% increase in ordinary dividends and a 23% increase in total compensation expense. The return on equity was 18.7% and the effective tax rate was 25.2%. The company also saw a 26% increase in CEO awarded profit share and a 16% increase in total Executive KMP awarded profit share.

Document Similarity Search

 **Finance GPT**

Explore financial documents with the power of GPT.

Input your prompt here

What are the main financial highlights from the latest annual report?

Enter a prompt related to finance to get insights from your document collection.

The main financial highlights from the latest annual report include a 22% increase in total operating expenses, a 76% increase in income tax expense, and a 56% increase in NPAT (Net Profit After Tax). Additionally, there was a 32% increase in ordinary dividends and a 23% increase in total compensation expense. The return on equity was 18.7% and the effective tax rate was 25.2%. The company also saw a 26% increase in CEO awarded profit share and a 16% increase in total Executive KMP awarded profit share.

Document Similarity Search

Asynchronous Semantic Search Integration for Enhanced Query Processing

```
if overrides.get("semantic_ranker") and has_text:
    r = await self.search_client.search(query_text,
                                       filter=filter,
                                       query_type=QueryType.SEMANTIC,
                                       query_language="en-us",
                                       query_speller="lexicon",
                                       semantic_configuration_name="default",
                                       top=top,
                                       query_caption="extractive|highlight-false" if use_semantic_captions else None,
                                       vector=query_vector,
                                       top_k=50 if query_vector else None,
                                       vector_fields="embedding" if query_vector else None)
else:
    r = await self.search_client.search(query_text,
                                       filter=filter,
                                       top=top,
                                       vector=query_vector,
                                       top_k=50 if query_vector else None,
                                       vector_fields="embedding" if query_vector else None)

if use_semantic_captions:
    results = [doc[self.sourcepage_field] + ": " + nonewlines(" ".join([c.text for c in doc["@search.captions"]])) async for doc in r]
else:
    results = [doc[self.sourcepage_field] + ": " + nonewlines(doc[self.content_field]) async for doc in r]
content = "\n".join(results)
```

Semantic Ranker Check:

The code starts by checking if the `semantic_ranker` option is enabled (`overrides.get("semantic_ranker")`) and if there is text content to work with (`has_text`). These conditions determine whether to use a semantic ranking approach or a standard search query.

Semantic Search Execution:

If semantic ranking is applicable, the code constructs a semantic search query using the `search_client`. The query includes:

Query Text: The main search string input (`query_text`).

Filter: Any filters applied to refine search results.

Query Type: Specified as `SEMANTIC` to enable semantic search.

Query Language: The language of the query, here set to English (`en-us`).

Query Speller: A lexicon-based spell checker.

Semantic Configuration: Configurations related to the semantic search.

Top: The number of results to return.

Query Caption: Optional setting for how results should be captioned.

Vector and Vector Fields: These are used if the query also includes vector-based search parameters (like embeddings).

Standard Search Execution:

If semantic ranking is not enabled or there is no text, the standard search is performed without semantic enhancements. The parameters are similar, but it doesn't specify the QueryType as SEMANTIC and lacks some semantic-specific settings like `semantic_configuration_name`.

Results Handling:

Depending on the `use_semantic_captions` setting, the code either:

Formats the results to include captions if available (`doc['@search.captions']`), concatenating them with the document's main content field. This involves extracting text from captions, handling newline characters to format the output neatly.

Simply formats results using a specific content field from each document (`doc[self.content_field]`).

Content Aggregation:

The results (whether they include captions or not) are aggregated into a single string with each result on a new line. This makes the output easy to read or display in a user interface or report.

Iterative Chatbot Query Handling and Response Generation with Semantic Search Integration

```
user_input = "Does my plan cover annual eye exams?"
exclude_category = None
if len(history) > 0:
    completion = openai.Completion.create(
        engine=AZURE_OPENAI_GPT_DEPLOYMENT,
        prompt=summary_prompt_template.format(summary="\n".join(history), question=user_input),
        temperature=0.7,
        max_tokens=32,
        stop=["\n"])
    search = completion.choices[0].text
else:
    search = user_input
print("Searching:", search)
print("-----")
filter = "category ne '{}'.format(exclude_category.replace("'", "")) if exclude_category else None
r = search_client.search(search,
    filter=filter,
    query_type=QueryType.SEMANTIC,
    query_language="en-us",
    query_speller="lexicon",
    semantic_configuration_name="default",
    top=3)
results = [doc[KB_FIELDS_SOURCEPAGE] + ": " + doc[KB_FIELDS_CONTENT].replace("\n", "").replace("\r", "") for doc in r]
content = "\n".join(results)
prompt = prompt_prefix.format(sources=content) + prompt_history + user_input + turn_suffix

completion = openai.Completion.create(
    engine=AZURE_OPENAI_CHATGPT_DEPLOYMENT,
    prompt=prompt,
    temperature=0.7,
    max_tokens=1024,
    stop=["<|im_end|>", "<|im_start|>"])
prompt_history += user_input + turn_suffix + completion.choices[0].text + "\n<|im_end|>" + turn_prefix
history.append("user: " + user_input)
history.append("assistant: " + completion.choices[0].text)

print("\n-----\n".join(history))
print("\n-----\nPrompt:\n" + prompt)
```

User Input Handling:

The `user_input` variable captures the user's query. In this case, the query is about whether their plan covers annual eye exams.

Chat History Accumulation:

The script checks if there is any existing chat history (`history` list). If there is, it uses this history to generate a new query prompt, which helps the model maintain context over the conversation.

Query Generation:

If there is history, the script constructs a new prompt combining the history with the current `user_input`, and requests a completion from the GPT model (`openai.Completion.create`). The

completion presumably refines or reformulates the query for better search results or for following up on the conversation.

If there is no history, it directly uses the `user_input` as the search query.

Semantic Search Execution:

The script formats a search query and executes it against a search client (`search_client.search`). It uses semantic search parameters to retrieve relevant documents from a knowledge base. The search is filtered to exclude certain categories if specified (`exclude_category`).

The results are then formatted to remove newline and carriage return characters, making them cleaner for display or further processing.

Response Generation:

Using the results from the semantic search, a new prompt is constructed and sent to another instance of GPT (possibly a more conversationally-focused model) to generate a text completion that forms the chatbot's response to the user query.

The script manages conversation history by appending the user's input and the chatbot's response to the history list and updating the `prompt_history` to maintain the flow of the conversation.

Output:

The chat history and the current prompt are printed to the console, providing a running transcript of the conversation.

Chat History Update:

The history of the chat (both user inputs and bot responses) is continually updated after each interaction. This allows the model to maintain context and provide coherent responses throughout the session.