Assignment 3-short: Agents with Memory Recall via RAG

Student ID: 111502026, Name: 薛耀智

1. (20%) Document Processing — Preparing Operation Manuals for Retrieval-Augmented Generation Systems

How can operation manuals in various formats (e.g., PDF documents, scanned images, web pages) be processed and structured for use in a RAG system? Please describe how you would:

- Extract and convert text from different formats (e.g., PDF to Markdown, OCR for images)
- Apply chunking strategies to ensure optimal retrieval granularity
- Build a vector database using embeddings suitable for downstream retrieval
- Handle metadata (e.g., section titles, timestamps) to improve retrieval relevance and ranking

Following Answer Based on LangGraph Framework

1. Extracting and Converting Text from Multiple Formats

The first step is to convert knowledge from operation manuals into retrievable content, extracting text based on the source format. For PDF files, tools such as pdfplumber or PyMuPDF can be used to read content and convert it into structured Markdown format. If using LangChain, the UnstructuredPDFLoader is recommended for efficiently processing PDFs. For scanned images (such as photos or handwritten documents), OCR tools like Tesseract or Google Vision API can be used for text recognition, and this process can be integrated via LangChain's UnstructuredImageLoader. For web page content, one can use BeautifulSoup with readability-lxml to extract the main text body, or simply use LangChain's WebBaseLoader for parsing. Each of these capabilities can be encapsulated as tool nodes in LangGraph and exposed through a unified interface using the @tool decorator—for example, by implementing a load_pdf_to_md() tool to automatically load and convert PDF files to plain text.

2. Designing Effective Text Chunking Strategies for Retrieval

Since operation manuals are often lengthy, they must be appropriately segmented to enable efficient vector indexing. It is recommended to use structure- and semantics-aware chunking strategies—for example, splitting by section headers or paragraphs, limiting each chunk to 500 tokens, and preserving hierarchical section information. This process can be implemented using RecursiveCharacterTextSplitter, aligned with normalized formatting such as Markdown heading levels. In LangGraph, this logic can be designed as a split_text_into_chunks() tool node that receives raw text and returns a list of chunks, each optionally carrying source metadata such as section title or page number, for use in downstream embedding and retrieval stages.

3. Building a Vector Database to Support Semantic Retrieval

After chunking, each segment needs to be embedded into vector representations to support semantic similarity search. Embedding models such as OpenAIEmbeddings, HuggingFaceEmbeddings, or InstructorEmbeddings can be used to convert text into vectors, which are then stored in a retrievable vector database such as FAISS, Chroma, or Weaviate.

This stage can be implemented in LangGraph through an embed_and_store() tool node, which wraps each text chunk along with its metadata into a Document object, embeds them in batch, and writes them into the vector store. This vector database can later be queried directly by the RAG system, enabling efficient document recall and enhanced context generation.

4. Integrating and Leveraging Metadata to Improve Retrieval Quality

During the vector database construction, preserving metadata for each text chunk is crucial to improving retrieval quality. Recommended metadata fields include section titles (e.g., "2-1 Creating an Account"), page numbers or timestamps, source type (PDF, image, or HTML), and file name. These metadata fields can be used for query filtering, semantic ranking, and content traceability. For example, you may filter results to only include sections with specific titles or prioritize image-based segments. In LangGraph, each Document can carry a metadata dictionary, and in later stages, such as similarity search, metadata filters can be applied using methods like .similarity_search() with conditional parameters (e.g., section title contains keyword) to boost retrieval accuracy.

2. (25%) Prompt Engineering — Designing Effective Prompts for RAG-based Browser Agents

How can prompt engineering be used to enhance a RAG-based system that helps browser agents complete complex tasks?
Please explain:
- How to design system-level prompts to guide the model's overall behavior (e.g., role, constraints, language use)
- How to create task-specific prompts that adapt to different user goals and input contexts
- How to integrate retrieved manual content into prompts effectively (e.g., in-context examples, instruction chaining)
- How to manage prompt length, relevance ranking, and hallucination avoidance when working with long manuals or multi-step procedures

1. Designing System-Level Prompts to Guide Overall Model Behavior

In a RAG-based system, system-level prompts play a crucial role in establishing the overarching behavioral norms of the model. These prompts are typically placed at the initial nodes of LangGraph and are responsible for assigning a clear role to the model (e.g., intelligent browser assistant), defining language and output formats (such as Chinese, JSON commands, or step-by-step instructions), and setting boundary constraints (e.g., the model should not fabricate steps that don’t exist and must verify the current page state before proceeding). This kind of prompt design ensures the model operates within a reliable and controllable scope throughout the task execution. For example, a well-crafted system prompt might say: "You are a professional browser assistant that helps users interact with websites. Your task is to follow the operation manual to complete complex web-based workflows, such as filling out forms or creating accounts. You must act cautiously and step-by-step, verifying prerequisites before each action and never imagining steps not found in the manual."

2. Dynamically Generating Task-Specific Prompts Based on User Goals

To handle varying user needs and task objectives, a RAG system must be capable of generating prompts dynamically. First, intent recognition can be applied to extract specific task goals from user input (e.g., “I want to check my application status”). Then, a matching prompt template can be selected based on task type—for example, prompts for query tasks might guide the model to inspect table fields or look for status updates, while download tasks might involve button clicking or link interaction flows. Additionally, contextual information such

as the user's current page, interface language, or preferences can be injected into the prompt to enhance its relevance. These task prompts can be modularized in LangGraph using a PromptBuilder node, allowing different templates to be automatically composed and adapted in real time based on context.

3. Effectively Integrating Retrieved Manual Content into Prompts

The key advantage of a RAG system lies in its ability to incorporate external knowledge, making it vital to embed retrieved manual content into prompts effectively. One recommended approach is "instruction chaining," where the relevant manual snippet is first presented, followed by a directive asking the model to plan actions based on it, e.g., "Based on the following manual content, please outline the full procedure for account creation: ...". Another strategy involves few-shot in-context examples, providing similar past task prompts and responses to reinforce understanding. Additionally, it's beneficial to annotate the source of each snippet (e.g., section title, page number) to increase the model's trust in the information and facilitate later verification. These integration strategies can be implemented using LangChain's composition strategies like stuff, map-reduce, or refine, in combination with LangGraph nodes.

4. Managing Prompt Length, Content Ranking, and Hallucination Avoidance

Due to the complexity and length of operation manuals, improper handling can lead to overly long prompts, irrelevant content, or even hallucinated responses. To prevent this, it's important to first filter for segments most relevant to the current task, summarize or extract procedural steps if needed, and set a maximum token limit per chunk. Retrieved content should then be ranked by semantic similarity (top-k), possibly incorporating the user's current webpage context as an additional weighting factor to refine relevance. To minimize hallucination risks, prompts should include warnings such as: "Only respond based on the manual content. If the information is insufficient, clearly state that the answer cannot be determined." The model should be explicitly restricted from inventing steps not present in the source. These safeguards can be implemented in LangGraph nodes using conditional filters and validation logic.

3. (25%) RAG — System Architecture and Query-driven Retrieval for Manual-Guided Task Completion

How can a Retrieval-Augmented Generation (RAG) system be designed to help browser agents complete unfamiliar tasks by leveraging retrieved operation manuals?
Please discuss:

- The end-to-end RAG architecture, from query understanding to retrieval and final generation
- How the system retrieves relevant document chunks based on the current task or the agent's intermediate intent (i.e., what the agent is currently reasoning about or attempting to do)
- Optional post-processing techniques for retrieved content (e.g., reranking, filtering, summarization) to enhance precision
- How to incorporate the retrieved content into the generation process to derive accurate and actionable step-by-step instructions
- Optional design consideration: In RAG-guided interactive browsing tasks (ref Slide 19 or 22), could the assistant's responses be enhanced by including features such as a Step Tracker (e.g., "Step 4 of 9") or an Instruction Cue (e.g., "Now focus on filtering by date")? Discuss whether such elements may help the agent better execute actions, stay aligned with

the task flow, and focus on the next appropriate action. Feel free to propose additional interaction design ideas that could improve clarity and guidance for the agent.

You may include examples of previously unachievable tasks and explain how the agent successfully completed them after referencing retrieved instructions.

1. The end-to-end RAG architecture, from query understanding to retrieval and final generation

In this movie-knowledge-driven MAS system, I initially adopted a traditional three-stage agent workflow: plan → execute → replan. The planning agent would create a task plan based on the user's input, the execution agent would carry out the actions, and if deviations occurred during execution, the replan agent would revise the plan accordingly. However, I soon realized that for similar or repetitive tasks—such as only changing the main character or asking about different metadata—the system would redundantly replan each time, leading to inefficiencies. To address this, I designed a vector-based Retrieval-Augmented Generation (RAG) architecture that enables the system to "remember" previously executed tasks. In this setup, the user's query is embedded into a vector and matched against a vectorstore containing past queries and their corresponding plans. If the similarity score exceeds a threshold, the system can skip the planning phase and resume directly from the previously saved plan, continuing with the execute → replan stages. Otherwise, it follows the standard full cycle. This design gives my system memory and reuse capability, significantly improving efficiency and avoiding unnecessary reasoning.

2. How the system retrieves relevant document chunks based on the current task or the agent's intermediate intent

My retrieval mechanism uses the user input as the query, while the vectorstore stores pairs of previous inputs and their corresponding task plans. This setup enables the agent to align its current reasoning state with previously successful tasks. For example, if the agent detects that the user’s query involves virtual reality and Keanu Reeves, it can retrieve and apply the process previously used to identify The Matrix. This retrieval is not limited to surface-level matching but also leaves room for future expansion into multi-stage retrieval—starting with task-level matching and drilling down into subtask-level similarity (e.g., checking for similar queries related specifically to "director lookup").

3. Optional post-processing techniques for retrieved content (e.g., reranking, filtering, summarization) to enhance precision

To improve retrieval accuracy and task alignment, I apply a simplified post-processing step after retrieving candidate plans. When multiple similar records are returned from the vectorstore, I simply select the one with the highest similarity score as the reusable plan. This allows the system to efficiently identify the most semantically relevant match without introducing latency from further reranking or filtering. While I haven’t yet incorporated additional modules like semantic reranking or content summarization, I’ve intentionally left room for future expansion. For example, I may later introduce filters to exclude failed tasks or extract key steps from long plans. The goal of this approach is to maintain a lightweight system while ensuring precision where it matters most—at the core of task planning and reuse.

4. How to incorporate the retrieved content into the generation process to derive accurate and actionable step-by-step instructions

Once I retrieve the most relevant plan from the vectorstore, I pass it directly to the execute agent to carry out the next phase of the task, skipping any additional prompt rephrasing or

narrative injection. This enables the system to bypass replanning entirely and proceed immediately with task execution, significantly improving responsiveness and runtime efficiency. However, to preserve flexibility and allow for corrections, I still retain the execute → replan mechanism. If the execution fails or the current task context changes, the system can fall back and adjust accordingly. This design ensures that while past plans are reused when possible, the system remains adaptive and responsive to the real-time task flow.

5. Optional design consideration: Enhancing interaction with memory, step tracking, and instruction cues

In terms of interaction design, I've added a memory mechanism that allows the system to persist task state across multiple user inputs. After executing a user command and returning a result, the system retains awareness of the ongoing task. If the user expresses dissatisfaction or provides follow-up corrections, the system can adjust its actions accordingly based on the previously stored plan and context. This memory capability allows the agent to support multi-turn task understanding without reprocessing the original goal from scratch. In addition, I've designed features like a step tracker (e.g., "Step 4 of 9") to help users and the agent align on task progress, and instruction cues (e.g., "Now focus on filtering by date") to direct the agent's attention to the current subtask. These enhancements not only improve the fluency of task execution but also increase the system's flexibility, accuracy, and user trust in multi-step reasoning scenarios.