QA and Chat over Documents | Langchain

Skip to main content LangChainDocsUse casesIntegrationsAPIMoreCommunityTutorialsContributingAlso by LangChainChat our docsLangSmithLangChain HubLangServePython DocsSearchCTRLKUse casesQA and Chat over DocumentsAdvanced Conversational QAConversational Retrieval AgentsUse local LLMsRetrieval-augmented generation (RAG)Tabular Question AnsweringInteracting with APIsSummarizationAgent SimulationsAutonomous AgentsChatbotsExtractionUse casesQA and Chat over DocumentsQA and Chat over DocumentsChat and Question-Answering (QA) over data are popular LLM use-cases.data can include many things, including:Unstructured data (e.g., PDFs)Structured data (e.g., SQL)Code (e.g., Python)Below we will review Chat and QA on Unstructured data.Unstructured data can be loaded from many sources.Check out the document loader integrations here to browse the set of supported loaders.Each loader returns data as a LangChain Document.Documents are turned into a Chat or QA app following the general steps below:Splitting: Text splitters break Documents into splits of specified sizeStorage: Storage (e.g., often a vectorstore) will house and often embed the splitsRetrieval: The app retrieves splits from storage (e.g., often with similar embeddings to the input question)Output: An LLM produces an answer using a prompt that includes the question and the retrieved splitsQuickstartLet's load this blog post on agents as an example Document.We'll have a QA app in a few lines of code.First, set environment variables and install packages required for the guide:> yarn add cheerio# Or load env vars in your preferred way:> export OPENAI_API_KEY="..."1. Loading, Splitting, Storage1.1 Getting startedSpecify a Document loader.// Document loaderimport { CheerioWebBaseLoader } from "langchain/document_loaders/web/cheerio";const loader = new CheerioWebBaseLoader("https://lilianweng.github.io/posts/2023-06-23-agent/");const data = await loader.load();Split the Document into chunks for embedding and vector storage.import { RecursiveCharacterTextSplitter } from "langchain/text_splitter";const textSplitter = new RecursiveCharacterTextSplitter({ chunkSize: 500, chunkOverlap: 0,});const splitDocs = await textSplitter.splitDocuments(data);Embed and store the splits in a vector database (for demo purposes we use an unoptimized, in-memory example but you can browse integrations here):import { OpenAIEmbeddings } from "langchain/embeddings/openai";import { MemoryVectorStore } from "langchain/vectorstores/memory";const embeddings = new OpenAIEmbeddings();const vectorStore = await MemoryVectorStore.fromDocuments(splitDocs, embeddings);Here are the three pieces together:1.2 Going Deeper1.2.1 IntegrationsDocument LoadersBrowse document loader integrations here.See further documentation on loaders here.Document TransformersAll can ingest loaded Documents and process them (e.g., split).See further documentation on transformers here.VectorstoresBrowse vectorstore integrations here.See further documentation on vectorstores here.2. Retrieval2.1 Getting startedRetrieve relevant splits for any question using similarity_search.const relevantDocs = await vectorStore.similaritySearch("What is task decomposition?");console.log(relevantDocs.length);// 42.2 Going Deeper2.2.1 RetrievalVectorstores are commonly used for retrieval.But, they are not the only option.For example, SVMs (see thread here) can also be used.LangChain has many retrievers and retrieval methods including, but not limited to, vectorstores.All retrievers implement some common methods, such as getRelevantDocuments().3. QA3.1 Getting startedDistill the retrieved documents into an answer using an LLM (e.g., gpt-3.5-turbo) with RetrievalQA chain.import { RetrievalQAChain } from "langchain/chains";import { ChatOpenAI } from "langchain/chat_models/openai";const model = new ChatOpenAI({ modelName: "gpt-3.5-turbo" });const chain = RetrievalQAChain.fromLLM(model, vectorStore.asRetriever());const response = await chain.call({ query: "What is task decomposition?",});console.log(response);/* { text: 'Task decomposition refers to the process of breaking down a larger task into smaller, more manageable subgoals. By decomposing a task, it becomes easier for an agent or system to handle complex tasks efficiently. Task decomposition can be done through various methods such as using prompting or task-specific instructions, or through human inputs. It helps in planning and organizing the steps required to complete a task effectively.' }*/3.2 Going Deeper3.2.1 IntegrationsLLMsBrowse LLM integrations and further documentation here.3.2.2 Customizing the promptThe prompt in RetrievalQA chain can be customized as follows.import { RetrievalQAChain } from "langchain/chains";import { ChatOpenAI } from "langchain/chat_models/openai";import { PromptTemplate } from "langchain/prompts";const model = new ChatOpenAI({ modelName: "gpt-3.5-turbo" });const template = `Use the following pieces of context to answer the question at the end.If you don't know the answer, just say that you don't know, don't try to make up an answer.Use three sentences maximum and keep the answer as concise as possible.Always say "thanks for asking!" at the end of the answer.{context}Question: {question}Helpful Answer:`;const chain = RetrievalQAChain.fromLLM(model, vectorStore.asRetriever(), { prompt: PromptTemplate.fromTemplate(template),});const response = await chain.call({ query: "What is task decomposition?",});console.log(response);/* { text: 'Task decomposition is the process of breaking down a large task into smaller, more manageable subgoals. This allows for efficient handling of complex tasks and aids in planning and organizing the steps needed to achieve the overall goal. Thanks for asking!' }*/3.2.3 Returning source documentsThe full set of retrieved documents used for answer distillation can be returned using return_source_documents=True.import { RetrievalQAChain } from "langchain/chains";import { ChatOpenAI } from "langchain/chat_models/openai";const model = new ChatOpenAI({ modelName: "gpt-3.5-turbo" });const chain = RetrievalQAChain.fromLLM(model, vectorStore.asRetriever(), { returnSourceDocuments: true,});const response = await chain.call({ query: "What is task decomposition?",});console.log(response.sourceDocuments[0]);/*Document { pageContent: 'Task decomposition can be done (1) by LLM with simple prompting like "Steps for XYZ.\\n1.", "What are the subgoals for achieving XYZ?", (2) by using task-specific instructions; e.g. "Write a story outline." for writing a novel, or (3) with human inputs.', metadata: [Object]}*/3.2.4 Customizing retrieved docs in the LLM promptRetrieved documents can be fed to an LLM for answer distillation in a few different ways.stuff, refine, and map-reduce chains for passing documents to an LLM prompt are well summarized here.stuff is commonly used because it simply "stuffs" all retrieved documents into the prompt.The loadQAChain methods are easy ways to pass documents to an LLM using these various approaches.import { loadQAStuffChain } from "langchain/chains";const stuffChain = loadQAStuffChain(model);const stuffResult = await stuffChain.call({ input_documents: relevantDocs, question: "What is task decomposition?",});console.log(stuffResult);/*{ text: 'Task decomposition is the process of breaking down a large task into smaller, more manageable subgoals or steps. This allows for efficient handling of complex tasks by focusing on one subgoal at a time. Task decomposition can be done through various methods such as using simple prompting, task-specific instructions, or human inputs.'}*/4. Chat4.1 Getting startedTo keep chat history, we use a variant of the previous chain called a ConversationalRetrievalQAChain.

First, specify a Memory buffer to track the conversation inputs / outputs.import { ConversationalRetrievalQAChain } from "langchain/chains";import { BufferMemory } from "langchain/memory";import { ChatOpenAI } from "langchain/chat_models/openai";const memory = new BufferMemory({ memoryKey: "chat_history", returnMessages: true,});Next, we initialize and call the chain:const model = new ChatOpenAI({ modelName: "gpt-3.5-turbo" });const chain = ConversationalRetrievalQAChain.fromLLM(model, vectorStore.asRetriever(), { memory, });const result = await chain.call({ question: "What are some of the main ideas in self-reflection?",});console.log(result);/*{ text: 'Some main ideas in self-reflection include:\n' + '\n' + '1. Iterative Improvement: Self-reflection allows autonomous agents to improve by continuously refining past action decisions and correcting mistakes.\n' + '\n' + '2. Trial and Error: Self-reflection plays a crucial role in real-world tasks where trial and error are inevitable. It helps agents learn from failed trajectories and make adjustments for future actions.\n' + '\n' + '3. Constructive Criticism: Agents engage in constructive self-criticism of their big-picture behavior to identify areas for improvement.\n' + '\n' + '4. Decision and Strategy Refinement: Reflection on past decisions and strategies enables agents to refine their approach and make more informed choices.\n' + '\n' + '5. Efficiency and Optimization: Self-reflection encourages agents to be smart and efficient in their actions, aiming to complete tasks in the least number of steps.\n' + '\n' + 'These ideas highlight the importance of self-reflection in enhancing performance and guiding future actions.'}*/The Memory buffer has context to resolve "it" ("self-reflection") in the below question.const followupResult = await chain.call({ question: "How does the Reflexion paper handle it?",});console.log(followupResult);/*{ text: "The Reflexion paper introduces a framework that equips agents with dynamic memory and self-reflection capabilities to improve their reasoning skills. The approach involves showing the agent two-shot examples, where each example consists of a failed trajectory and an ideal reflection on how to guide future changes in the agent's plan. These reflections are then added to the agent's working memory as context for querying a language model. The agent uses this self-reflection information to make decisions on whether to start a new trial or continue with the current plan."}*/4.2 Going deeperThe documentation on ConversationalRetrievalQAChain offers a few extensions, such as streaming and source documents.PreviousUse casesNextAdvanced Conversational QACommunityDiscordTwitterGitHubPythonJS/TSMoreHomepageBlogCopyright © 2023 LangChain, Inc.