RAG over code | Langchain

Skip to main content LangChainDocsUse casesIntegrationsAPIMoreCommunityTutorialsContributingAlso by LangChainChat our docsLangSmithLangChain HubLangServePython DocsSearchCTRLKUse casesQA and Chat over DocumentsRetrieval-augmented generation (RAG)RAG over codeTabular Question AnsweringInteracting with APIsSummarizationAgent SimulationsAutonomous AgentsChatbotsExtractionUse casesRetrieval-augmented generation (RAG)RAG over codeOn this pageRAG over codeUse caseSource code analysis is one of the most popular LLM applications (e.g., GitHub Co-Pilot, Code Interpreter, Codium, and Codeium) for use-cases such as:Q&A over the code base to understand how it worksUsing LLMs for suggesting refactors or improvementsUsing LLMs for documenting the codeOverviewThe pipeline for QA over code follows the steps we do for document question answering, with some differences:In particular, we can employ a splitting strategy that does a few things:Keeps each top-level function and class in the code is loaded into separate documents.Puts remaining into a separate document.Retains metadata about where each split comes fromQuickstartyarn add @supabase/supabase-js# Set env var OPENAI_API_KEY or load from a .env fileLoadingWe'll upload all JavaScript/TypeScript files using the DirectoryLoader and TextLoader classes.The following script iterates over the files in the LangChain repository and loads every .ts file (a.k.a. documents):import { DirectoryLoader } from "langchain/document_loaders/fs/directory";import { TextLoader } from "langchain/document_loaders/fs/text";import { RecursiveCharacterTextSplitter } from "langchain/text_splitter";// Define the path to the repo to preform RAG on.const REPO_PATH = "/tmp/test_repo";We load the code by passing the directory path to DirectoryLoader, which will load all files with .ts extensions.

These files are then passed to a TextLoader which will return the contents of the file as a string.const loader = new DirectoryLoader(REPO_PATH, { ".ts": (path) => new TextLoader(path),});const docs = await loader.load();Next, we can create a RecursiveCharacterTextSplitter to split our code.We'll call the static fromLanguage method to create a splitter that knows how to split JavaScript/TypeScript code.const javascriptSplitter = RecursiveCharacterTextSplitter.fromLanguage("js", { chunkSize: 2000, chunkOverlap: 200,});const texts = await javascriptSplitter.splitDocuments(docs);console.log("Loaded ", texts.length, " documents.");Loaded 3324 documents.RetrievalQAWe need to store the documents in a way we can semantically search for their content.The most common approach is to embed the contents of each document then store the embedding and document in a vector store.When setting up the vector store retriever:We test max marginal relevance for retrievalAnd 5 documents returnedIn this example we'll be using Supabase, however you can pick any vector store with MMR search you'd like from our large list of integrations.import { createClient } from "@supabase/supabase-js";import { OpenAIEmbeddings } from "langchain/embeddings/openai";import { SupabaseVectorStore } from "langchain/vectorstores/supabase";const privateKey = process.env.SUPABASE_PRIVATE_KEY;if (!privateKey) throw new Error(`Expected env var SUPABASE_PRIVATE_KEY`);const url = process.env.SUPABASE_URL;if (!url) throw new Error(`Expected env var SUPABASE_URL`);const client = createClient(url, privateKey);Once we've initialized our client we can pass it, along with some more options to the .fromDocuments method on SupabaseVectorStore.For more instructions on how to set up Supabase, see the Supabase docs.const vectorStore = await SupabaseVectorStore.fromDocuments(texts, new OpenAIEmbeddings(), { client, tableName: "documents", queryName: "match_documents", });const retriever = vectorStore.asRetriever({ searchType: "mmr", // Use max marginal relevance search searchKwargs: { fetchK: 5 },});ChatWe'll setup our model and memory system just as we'd do for any other chatbot application.import { ChatOpenAI } from "langchain/chat_models/openai";Pipe the StringOutputParser through since both chains which use this model will also need this output parser.const model = new ChatOpenAI({ modelName: "gpt-4" }).pipe(new StringOutputParser());We're going to use BufferMemory as our memory chain. All this will do is take in inputs/outputs from the LLM and store them in memory.import { BufferMemory } from "langchain/memory";const memory = new BufferMemory({ returnMessages: true, // Return stored messages as instances of `BaseMessage` memoryKey: "chat_history", // This must match up with our prompt template input variable.});Now we can construct our main sequence of chains. We're going to be building ConversationalRetrievalChain using Expression Language.import { ChatPromptTemplate, MessagesPlaceholder, AIMessagePromptTemplate, HumanMessagePromptTemplate,} from "langchain/prompts";import { RunnableSequence } from "langchain/schema/runnable";import { formatDocumentsAsString } from "langchain/util/document";import { BaseMessage } from "langchain/schema";import { StringOutputParser } from "langchain/schema/output_parser";Construct the chainThe meat of this code understanding example will be inside a single RunnableSequence chain.

Here, we'll have a single input parameter for the question, and preform retrieval for context and chat history (if available).

Then we'll preform the first LLM call to rephrase the users question.

Finally, using the rephrased question, context and chat history we'll query the LLM to generate the final answer which we'll return to the user.PromptsStep one is to define our prompts. We need two, the first we'll use to rephrase the users question, the second we'll use to combine the documents and question.Question generator prompt:const questionGeneratorTemplate = ChatPromptTemplate.fromMessages([AIMessagePromptTemplate.fromTemplate("Given the following conversation about a codebase and a follow up question, rephrase the follow up question to be a standalone question."), new MessagesPlaceholder("chat_history"), AIMessagePromptTemplate.fromTemplate(`Follow Up Input: {question}Standalone question:`),]);Combine documents prompt:const combineDocumentsPrompt = ChatPromptTemplate.fromMessages([AIMessagePromptTemplate.fromTemplate("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.\n\n{context}\n\n"), new MessagesPlaceholder("chat_history"), HumanMessagePromptTemplate.fromTemplate("Question: {question}"),]);Next, we'll construct both chains.const combineDocumentsChain = RunnableSequence.from([{ question: (output: string) => output, chat_history: async () => { const { chat_history } = await memory.loadMemoryVariables({}); return chat_history; }, context: async (output: string) => { const relevantDocs = await retriever.getRelevantDocuments(output); return formatDocumentsAsString(relevantDocs); }, }, combineDocumentsPrompt, model, new StringOutputParser(),]);const conversationalQaChain = RunnableSequence.from([{ question: (i: { question: string }) => i.question, chat_history: async () => { const { chat_history } = await memory.loadMemoryVariables({}); return chat_history; }, }, questionGeneratorTemplate, model, new StringOutputParser(), combineDocumentsChain,]);These two are somewhat complex chain so let's break it down.First, we define our single input parameter: question: string. Below this we also define chat_history which is not sourced from the user's input, but rather preforms a chat memory lookup.Next, we pipe those variables through to our prompt, model and lastly an output parser. This first part will rephrase the question, and return a single string of the rephrased question.In the next block we call the combineDocumentsChain which takes in the output from the first part of the conversationalQaChain and pipes it through to the next prompt. We also preform a retrieval call to get the relevant documents for the question and any chat history which might be present.Finally, the RunnableSequence returns the result of the model & output parser call from the combineDocumentsChain. This will return the final answer as a string to the user.The last step is to invoke our chain!const question = "How can I initialize a ReAct agent?";const result = await conversationalQaChain.invoke({ question,});This is also where we'd save the LLM response to memory for future context.await memory.saveContext({   input: question,   },   {   output: result,   });console.log(result);/**The steps to initialize a ReAct agent are:1. Import the necessary modules from their respective packages.   \```   import { initializeAgentExecutorWithOptions } from "langchain/agents";   import { OpenAI } from "langchain/llms/openai";   import { SerpAPI } from "langchain/tools";   import { Calculator } from "langchain/tools/calculator";   \```2. Create instances of the needed tools (i.e., OpenAI model, SerpAPI, and Calculator), providing the necessary options.   \```   const model = new OpenAI({ temperature: 0 });   const tools = [new SerpAPI(process.env.SERPAPI_API_KEY, { location: "Austin,Texas,United States",   hl: "en",   gl: "us",   }),   new Calculator(),   ]; \```3. Call `initializeAgentExecutorWithOptions` function with the created tools and model, and with the desired options to create an agent instance.   \```   const executor = await initializeAgentExecutorWithOptions(tools, model, {   agentType: "zero-shot-react-description", });   \```Note: In async environments, the steps can be wrapped in a try-catch block to handle any exceptions that might occur during execution. As shown in some of the examples, the process can be aborted using an AbortController to cancel the process after a certain period of time for fail-safe reasons. */See the LangSmith trace for these two chains hereNext stepsSince we're saving our inputs/outputs in memory we can ask followups to the LLM.Keep in mind, we're not implementing an agent with tools so it must derive answers from the relevant documents in our store. Because of this it may return answers with hallucinated imports, classes or more.const question2 = "How can I import and use the Wikipedia and File System tools from LangChain instead?";const result2 = await conversationalQaChain.invoke({ question: question2,});console.log(result2);/**Here is how to import and utilize WikipediaQueryRun and File System tools in the LangChain codebase:1. First, you have to import necessary tools and classes:\```javascript// for file system toolsimport { ReadFileTool, WriteFileTool, NodeFileStore } from "langchain/tools";// for wikipedia toolsimport { WikipediaQueryRun } from "langchain/tools";\```2. To use File System tools, you need an instance of File Store, either `NodeFileStore` for the server-side file system or `InMemoryFileStore` for in-memory file storage:\```javascript// example of instancing NodeFileStore for server-side file systemconst store = new NodeFileStore();\```3. Then you can instantiate your file system tools with this store:\```javascriptconst tools = [new ReadFileTool({ store }), new WriteFileTool({ store })];\```4. To use WikipediaQueryRun tool, first you have to instance it like this:\```javascriptconst wikipediaTool = new WikipediaQueryRun({ topKResults: 3, maxDocContentLength: 4000,});\```5. After that, you can use the `call` method of the created instance for making queries. For example, to query the Wikipedia for "Langchain":\```javascriptconst res = await wikipediaTool.call("Langchain");console.log(res);\```Note: This example assumes you're running the code in an asynchronous context. For synchronous context, you may need to adjust the code accordingly.*/See the LangSmith trace for this run here