Tutorial: Ollama tool calling

Overview

Tool calling in Large Language Models (LLMs) is the ability of the LLM to interact with external tools, services, or APIs to perform tasks. This allows LLMs to extend their functionality, enhancing their ability to handle real-world tasks that may require access to external data, real-time information, or specific applications. When an LLM uses a web search tool, it can call the web to fetch real-time data that aren't available in the model's training data. Other types of tools might include Python for calculations, data analysis, or visualization, or calling a service endpoint for data. Tool calling can make a chatbot more dynamic and adaptable, allowing it to provide more accurate, relevant, and detailed responses based on live data or specialized tasks outside its immediate knowledge base. Popular frameworks for tool-calling include Langchain and now ollama.

Ollama is a platform that offers open-source, local AI models for use on personal devices so that users can run LLMs directly on their computers. Unlike a service like the OpenAI API, there's no need for an account since the model is on your local machine. Ollama focuses on privacy, performance, and ease of use, enabling users to access and interact with AI models without sending data to external servers. This can be particularly appealing for those concerned about data privacy or who want to avoid the reliance on external APIs. Ollama's platform is designed to be easy to set up and use, and it supports various models, giving users a range of tools for natural language processing, code generation, and other AI tasks directly on their own hardware. It is well suited to a tool calling architecture because it can access all the capabilities of a local environment including data, programs, and custom software.

In this tutorial you'll learn how to set up tool calling by using ollama to look through a local filesystem, a task which would be difficult to do with a remote LLM. Many ollama models are available for tool calling and building AI agents like Mistral and Llama 3.2, a full list is available at the ollama website. In this case we'll use IBM Granite 3.2 Dense which has tool support. The 2B and 8B models are text-only dense LLMs trained on designed to support tool-based use cases and for retrieval augmented generation (RAG), streamlining code generation, translation and bug fixing.

The notebook for this tutorial can be downloaded from Github here.

Step 1: Install Ollama

First you'll download ollama from https://ollama.com/download and install it for your operating system. On macOS this is done via a .dmg file, on Linux via a single shell command, and on Windows with an installer. You may need admin access on your machine in order to run the installer.

You can test that ollama is correctly installed by opening a terminal or command prompt and entering:
ollama -v
Powered by Granite
Powered by Granite

Step 2: Install libraries

Next, you'll add the initial imports. This demo will use the ollama python library to communicate with ollama and the pymupdf library to read PDF files in the file system.
!pip install pymupdf import ollama import os import pymupdf
Powered by Granite

Next you'll pull the model that you'll be using throughout this tutorial. This downloads the model weights from ollama to your local computer and stores them for use without needing to make any remote API calls later on.
!ollama pull granite3.2 !ollama pull granite3.2-vision
Powered by Granite

Step 3: Define the tools

Now you'll define the tools that the ollama tools instance will have access. Since the intent of the tools is to read files and look through images in the local file system, you'll create two python functions for each of those tools. The first is calledsearch_text_files and it takes a keyword to search for in the local files. For the purposes of this demo, the code only searches for files in a specific folder but it could be extended to include a second parameter that sets which folder the tool will search in.

You could use simple string matching to see whether the keyword is in the document but because ollama makes calling local llms easily,search_text_files will use Granite 3.2 to determine whether the keyword describes the document text. This is done by reading the document into a string calleddocument_text . The function then calls ollama.chat and prompts the model with the following: "Respond only 'yes' or 'no', do not add any additional information. Is the following text about " + keyword + "? " + document_text
Powered by Granite

If the model responds 'yes', then the function returns the name of the file that contains the keyword that the user indicated in the prompt. If none of the files seem to contain the information, then the function returns 'None' as a string.

This function may run slowly the first time because ollama will download Granite 3.2 Dense.
```
def search_text_files(keyword: str) -> str:  directory = os.listdir("./files/")  for fname in directory:  # look through all the files in our directory that aren't hidden files  if os.path.isfile("./files/" + fname) and not fname.startswith('.'):  if(fname.endswith(".pdf")):  document_text = ""  doc = pymupdf.open("./files/" + fname)  for page in doc: # iterate the document pages  document_text += page.get_text() # get plain text (is in UTF-8)  doc.close()  prompt = "Respond only 'yes' or 'no', do not add any additional information. Is the following text about " + keyword + "? " + document_text  res = ollama.chat(  model="granite3.2:8b",  messages=[{'role': 'user', 'content': prompt}]  )  if 'Yes' in res['message']['content']:  return "./files/" + fname  elif(fname.endswith(".txt")):  f = open("./files/" + fname, 'r')  file_content = f.read()  prompt = "Respond only 'yes' or 'no', do not add any additional information. Is the following text about " + keyword + "? " + file_content  res = ollama.chat(  model="granite3.2:8b",  messages=[{'role': 'user', 'content': prompt}]  )  if 'Yes' in res['message']['content']:  f.close()  return "./files/" + fname  return "None"
```
Powered by Granite

The second tool is called search_image_files and it takes a keyword to search for in the local photos.The search is done by using the Granite 3.2 Vision image description model via ollama. This model will return a text description of each image file in the folder and search for the keyword in the description. One of the strengths of using ollama is that multi-agent systems can easily be built to call one model with another.

The function returns a string, which is the name of the file whose description contains the keyword that the user indicated in the prompt.

```
def search_image_files(keyword:str) -> str:  directory = os.listdir("./files/")  image_file_types = ("jpg", "png", "jpeg")  for fname in directory:  if os.path.isfile("./files/" + fname) and not fname.startswith('.') and fname.endswith(image_file_types):  res = ollama.chat( model="granite3.2-vision",  messages=[  {  'role': 'user',  'content': 'Describe this image in short sentences. Use simple phrases first and then describe it more fully.',  'images': ["./files/" + fname]  }  ]  )  if keyword in res['message']['content']:  return "./files/" + fname  return "None"
```

Powered by Granite

Step 4: Define the tools for ollama

Now that the functions for ollama to call have been defined, you'll configure the tool information for ollama itself. The first step is to create an object that maps the name of the tool to the functions for ollama function calling:

```
available_functions = {  'Search inside text files':search_text_files,  'Search inside image files':search_image_files }
```

Powered by Granite

Next, configure a tools array to tell ollama what tools it will have access to and what those tools require. This is an array with one object schema per tool that tells the ollama tool calling framework how to call the tool and what it returns.

In the case of both of the tools that you created earlier, they are functions that require a keyword parameter. Currently only functions are supported although this may change in the future. The description of the function and of the parameter help the model call the tool correctly. The description field for the function of each tool is passed to the LLM when it selects which tool to use. The description of the keyword is passed to the model when it generates the parameters that will be passed to the tool. Both of these are places you may look to fine tune prompts when you create your own tool calling applications with ollama.

```
# tools don't need to be defined as an object but this helps pass the correct parameters # to the tool call itself by giving the model a prompt of how the tool is to be used ollama_tools=[  {  'type': 'function',  'function': {  'name': 'Search inside text files',  'description': 'This tool searches in PDF or plaintext or text files in the local file system for descriptions or mentions of the keyword.', 'parameters': {  'type': 'object',  'properties': {  'keyword': {  'type': 'string',  'description': 'Generate one keyword from the user request to search for in text files',  },  },  'required': ['keyword'],  },  },  },  {  'type': 'function',  'function': { 'name': 'Search inside image files',  'description': 'This tool searches for photos or image files in the local file system for the keyword.',  'parameters': {  'type': 'object',  'properties': { 'keyword': {  'type': 'string',  'description': 'Generate one keyword from the user request to search for in image files',  },  },  'required': ['keyword'],  },  },  },  ]
```

Powered by Granite

You'll use this tools definition when you call ollama with user input.

Step 5: Pass user input to ollama

Now its time to pass user input to ollama and have it return the results of the tool calls. First, make sure that ollama is running on your system:
if ollama is not currently running, start it import subprocess subprocess.Popen(["ollama","serve"], stdout=subprocess.DEVNULL, stderr=subprocess.STDOUT)
Powered by Granite