Build a Tool Calling Agent

Estimated time needed: **1** hour

In this lab, you'll explore the powerful capabilities of tool calling in large language models (LLMs) to build advanced AI agents that can interact with external systems. You'll learn how to create custom tools that enable an LLM to perform specific actions, from extracting video IDs to fetching YouTube transcripts and metadata. Through hands-on examples, you'll first implement manual tool calling to understand the underlying mechanics, then build a flexible YouTube interaction system that can search videos, extract transcripts, fetch trending content, and generate summaries. By the end of this lab, you'll understand how to construct both fixed-sequence and recursive tool-calling chains, allowing your AI assistants to dynamically decide which tools to use and when to use them, creating truly intelligent agents that can reason about and interact with the world around them.

Table of Contents

Objectives

After completing this lab you will be able to:

- Create custom tools that extend the capabilities of language models
- Build both manual and automated tool calling chains
- Implement recursive tool calling for dynamic, multi-step operations
- Develop AI agents that can interact with YouTube's content programmatically
- Apply tool calling techniques to extract, process, and summarize information from external sources
- Design flexible workflows that allow LLMs to reason about when and how to use available tools

---

Setup

For this lab, you will be using the following libraries:

- `pytube` for accessing YouTube videos and their metadata programmatically.
- `youtube-transcript-api` for fetching transcripts from YouTube videos.
- `langchain` for building tool-enabled LLM applications.
- `langchain-community` for additional LangChain integrations.
- `langchain-openai` for connecting to OpenAI's language models.
- `yt-dlp` for enhanced YouTube data extraction capabilities.

Installing required libraries

In [1]:
```
%%capture
%pip install pytube
%pip install youtube-transcript-api==1.1.0
%pip install langchain-community==0.3.16
%pip install langchain==0.3.23
%pip install langchain-openai==0.3.14
%pip install yt-dlp
```

Importing required libraries

It is recommended that you import all required libraries in one place (here):

In [2]:
```
import re
from pytube import YouTube
from langchain_core.tools import tool
from IPython.display import display, JSON
import yt_dlp
from typing import List, Dict
from langchain_core.messages import HumanMessage
from langchain_core.messages import ToolMessage
import json

# Suppress warnings
import warnings
warnings.filterwarnings("ignore")

# Suppress pytube errors
import logging
pytube_logger = logging.getLogger('pytube')
pytube_logger.setLevel(logging.ERROR)

# Suppress yt-dlp warnings
yt_dpl_logger = logging.getLogger('yt_dlp')
yt_dpl_logger.setLevel(logging.ERROR)
```

Let's initialize the language model that will power your tool calling capabilities. This code sets up a GPT-4o-mini model using the OpenAI provider through LangChain's interface, which you'll use to process queries and decide which tools to call.

In [3]:
```
from langchain.chat_models import init_chat_model

llm = init_chat_model("gpt-4o-mini", model_provider="openai")
```

API Disclaimer

This lab uses LLMs provided by OpenAI. This environment has been configured to allow LLM use without API keys so you can prompt them for **free (with limitations)**. With that in mind, if you wish to run this notebook **locally outside** of Skills Network's JupyterLab environment, you will have to configure your own API keys. Please note that using your own API keys means that you will incur personal charges.

Running Locally

If you are running this lab locally, you will need to configure your own API key. This lab uses the `init_chat_model` function from `langchain`. To use the model you must

set the environment variable `OPENAI_API_KEY` to your OpenAI API key. **DO NOT** run the cell below if you aren't running locally, it will causes errors.

In [5]:
```
# IGNORE IF YOU ARE NOT RUNNING LOCALLY
#os.environ["OPENAI_API_KEY"] = "your OpenAI API key here"
```

Tools

Creating custom tools with LangChain

Anatomy of a tool

Let's provide the basic building blooks a tool, consider the following tools:

```
@tool
def tool_name(input_param: input_type) -> output_type:
    """
    Clear description of what the tool does.

    Args:
        input_param (input_type): Description of this parameter

    Returns:
        output_type: Description of what is returned
    """
    # Function implementation
    result = process(input_param)
    return result
```

Key components

1. **@tool decorator**
    - Registers the function with LangChain
    - Creates tool attributes (.name, .description, .func)
    - Generates JSON schema for validation
    - Transforms regular functions into callable tools
2. **Function name**
    - Used by LLM to select appropriate tool
    - Used as reference in chains and tool mappings
    - Appears in tool call logs for debugging
    - Should clearly indicate the tool's purpose
3. **Type annotations**

- Enable automatic input validation
- Create schema for parameters
- Allow proper serialization of inputs/outputs
- Help LLM understand required input formats

4. **Docstring**

- Provides context for the LLM to decide when to use the tool
- Documents parameter requirements
- Explains expected outputs and behavior
- Critical for tool selection by the LLM

5. **Implementation**

- Executes the actual operation
- Handles errors appropriately
- Returns properly formatted results
- Should be efficient and robust

Defining video ID extraction tool

Now you'll define a function `extract_video_id` by denoting it as a tool that will help you to extract the video ID from a given URL. This is necessary because many YouTube API operations, including transcript extraction, require the video ID rather than the complete URL. The function uses regular expressions to handle different YouTube URL formats (standard, shortened, and embedded) and extract the 11-character video ID.

In [6]:
```python
@tool
def extract_video_id(url: str) -> str:
    """
    Extracts the 11-character YouTube video ID from a URL.

    Args:
        url (str): A YouTube URL containing a video ID.

    Returns:
        str: Extracted video ID or error message if parsing fails.
    """

    # Regex pattern to match video IDs
    pattern = r'(?:v=|be/|embed/)([a-zA-Z0-9_-]{11})'
    match = re.search(pattern, url)
    return match.group(1) if match else "Error: Invalid YouTube URL"
```

The decorator wraps your function, adding those attributes (.name, .description, .func) and registering it with LangChain's tool system. The original function becomes accessible through the .func attribute, but the overall object is an instance of LangChain's tool class, with additional methods like .run() for direct invocation.

Testing the video ID extraction tool

Now you'll be testing your `extract_video_id` tool to verify that it's correctly registered with LangChain. These print statements will show you:

1. The tool's name (as it will be referenced by the LLM)
2. The tool's description (which helps the LLM understand when to use this tool)
3. The actual function reference that will be called

In [7]:
```
print(extract_video_id.name)
print("----------------------------")
print(extract_video_id.description)
print("----------------------------")
print(extract_video_id.func)
```

```
extract_video_id
----------------------------
Extracts the 11-character YouTube video ID from a URL.

Args:
    url (str): A YouTube URL containing a video ID.

Returns:
    str: Extracted video ID or error message if parsing fails.
----------------------------
<function extract_video_id at 0x776f50cb16c0>
```

Testing tool execution

Here, you're testing the actual execution of your `extract_video_id` tool with a real YouTube URL. You can call the tool using the `.run()` method, which is a convenient way to execute the tool directly and see its output.

In [8]:
```
extract_video_id.run("https://www.youtube.com/watch?v=hfIUstzHs9A")
```

Out[8]:
```
'hfIUstzHs9A'
```

In [9]:
```
extract_video_id
```

Out[9]:
```
StructuredTool(name='extract_video_id', description='Extracts the 11-charac
ter YouTube video ID from a URL.\n\nArgs:\n    url (str): A YouTube URL con
taining a video ID.\n\nReturns:\n    str: Extracted video ID or error messa
ge if parsing fails.', args_schema=<class 'langchain_core.utils.pydantic.ex
tract_video_id'>, func=<function extract_video_id at 0x776f50cb16c0>)
```

This output shows that your function has been transformed into a `StructuredTool` object by LangChain. It displays the tool's name ('extract_video_id'), its description (our docstring), a Pydantic schema for input validation, and a reference to your original function.

Tool list

Multiple tools will be created to enhance the LLM's capabilities. For organization, create a list called tools, which is a standard Python list that contains tool objects created with the @tool decorator. This list doesn't execute functions or determine call order - it simply collects tool objects in one place so they can be efficiently passed to the language model via llm.bind_tools(tools). This approach allows the LLM to access all available tools without requiring them to be individually registered.

Adding the `extract_video_id` tool to your tools list, which you can later provide to the LLM so it can use this functionality when needed.

In [10]:
```
tools = []
tools.append(extract_video_id)
```

Now that you have understood the basic structure, let's define the rest of the tools you'll need.

Defining transcript fetching tool

Now you're going to create another tool that fetches the transcript from a YouTube video. This tool uses the `YouTubeTranscriptApi` library to retrieve the captions or subtitles from a video. You'll be taking the video ID (which can be extracted using your previous tool) and an optional language parameter. The function attempts to get the transcript and joins all text segments into a continuous string, or returns an error message if the transcript can't be retrieved.

In [11]:
```
from youtube_transcript_api import YouTubeTranscriptApi


@tool
def fetch_transcript(video_id: str, language: str = "en") -> str:
    """
    Fetches the transcript of a YouTube video.

    Args:
        video_id (str): The YouTube video ID (e.g., "dQw4w9WgXcQ").
        language (str): Language code for the transcript (e.g., "en", "es").

    Returns:
        str: The transcript text or an error message.
    """

    try:
        ytt_api = YouTubeTranscriptApi()
        transcript = ytt_api.fetch(video_id, languages=[language])
        return " ".join([snippet.text for snippet in transcript.snippets])
    except Exception as e:
        return f"Error: {str(e)}"
```

Let's test the fetch_transcript tool by directly calling it with the .run() method on a specific video ID. This will attempt to retrieve the transcript for the video with ID "hfIUstzHs9A" in the default English language.

In [12]:
```
fetch_transcript.run("hfIUstzHs9A")
```

Out[12]: 'Over the past couple of months, large language models, or LLMs, such as ch
atGPT, have taken the world by storm. Whether it\'s writing poetry or helpi
ng plan your upcoming vacation, we are seeing a step change in the performa
nce of AI and its potential to drive enterprise value. My name is Kate Soul
e. I\'m a senior manager of business strategy at IBM Research, and today I
\'m going to give a brief overview of this new field of AI that\'s emerging
and how it can be used in a business setting to drive value. Now, large lan
guage models are actually a part of a different class of models called foun
dation models. Now, the term "foundation models" was actually first coined
by a team from Stanford when they saw that the field of AI was converging t
o a new paradigm. Where before AI applications were being built by trainin
g, maybe a library of different AI models, where each AI model was trained
on very task-specific data to perform very specific task. They predicted th
at we were going to start moving to a new paradigm, where we would have a f
oundational capability, or a foundation model, that would drive all of thes
e same use cases and applications. So the same exact applications that we w
ere envisioning before with conventional AI, and the same model could drive
any number of additional applications. The point is that this model could b
e transferred to any number of tasks. What gives this model the super power
to be able to transfer to multiple different tasks and perform multiple dif
ferent functions is that it\'s been trained on a huge amount, in an unsuper
vised manner, on unstructured data. And what that means, in the language do
main, is basically I\'ll feed a bunch of sentences-- and I\'m talking terab
ytes of data here --to train this model. And the start of my sentence might
be "no use crying over spilled" and the end of my sentence might be "milk".
And I\'m trying to get my model to predict the last word of the sentence ba
sed off of the words that it saw before. And it\'s this generative capabili
ty of the model-- predicting and generating the next word --based off of pr
evious words that it\'s seen beforehand, that is why that foundation models
are actually a part of the field of AI called generative AI because we\'re
generating something new in this case, the next word in a sentence. And eve
n though these models are trained to perform, at its core, a generation pas
t, predicting the next word in the sentence, we actually can take these mod
els, and if you introduce a small amount of labeled data to the equation, y
ou can tune them to perform traditional NLP tasks-- things like classificat
ion, or named-entity recognition --things that you don\'t normally associat
e as being a generative-based model or capability. And this process is call
ed tuning. Where you can tune your foundation model by introducing a small
amount of data, you update the parameters of your model and now perform a v
ery specific natural language task. If you don\'t have data, or have only v
ery few data points, you can still take these foundation models and they ac
tually work very well in low-labeled data domains. And in a process called
prompting or prompt engineering, you can apply these models for some of tho
se same exact tasks. So an example of prompting a model to perform a classi
fication task might be you could give a model a sentence and then ask it a
question: Does this sentence have a positive sentiment or negative sentimen
t? The model\'s going to try and finish generating words in that sentence,
and the next natural word in that sentence would be the answer to your clas
sification problem, which would respond either positive or negative, depend
ing on where it estimated the sentiment of the sentence would be. And these
models work surprisingly well when applied to these new settings and domain
s. Now, this is a lot of where the advantages of foundation models come int
o play. So if we talk about the advantages, the chief advantage is the perf
ormance. These models have seen so much data. Again, data with a capital D-
- terabytes of data --that by the time that they\'re applied to small task
s, they can drastically outperform a model that was only trained on just a

```
few data points. The second advantage of these models are the productivity
gains. So just like I said earlier, through prompting or tuning, you need f
ar less label data to get to task-specific model than if you had to start f
rom scratch because your model is taking advantage of all the unlabeled dat
a that it saw in its pre-training when we created this generative task. Wit
h these advantages, there are also some disadvantages that are important to
keep in mind. And the first of those is the compute cost. So that penalty f
or having this model see so much data is that they\'re very expensive to tr
ain, making it difficult for smaller enterprises to train a foundation mode
l on their own. They\'re also expensive-- by the time they get to a huge si
ze, a couple billion parameters --they\'re also very expensive to run infer
ence. You might require multiple GPUs at a time just to host these models a
nd run inference, making them a more costly method than traditional approac
hes. The second disadvantage of these models is on the trustworthiness sid
e. So just like data is a huge advantage for these models, they\'ve seen so
much unstructured data, it also comes at a cost, especially in the domain l
ike language. A lot of these models are trained basically off of language d
ata that\'s been scraped from the Internet. And there\'s so much data that
these models have been trained on. Even if you had a whole team of human an
notators, you wouldn\'t be able to go through and actually vet every single
data point to make sure that it wasn\'t biased and didn\'t contain hate spe
ech or other toxic information. And that\'s just assuming you actually know
what the data is. Often we don\'t even know-- for a lot of these open sourc
e models that have been posted --what the exact datasets are that these mod
els have been trained on leading to trustworthiness issues. So IBM recogniz
es the huge potential of these technologies. But my partners in IBM Researc
h are working on multiple different innovations to try and improve also the
efficiency of these models and the trustworthiness and reliability of these
models to make them more relevant in a business setting. All of these examp
les that I\'ve talked through so far have just been on the language side. B
ut the reality is, there are a lot of other domains that foundation models
can be applied towards. Famously, we\'ve seen foundation models for vision
--looking at models such as DALL-E 2, which takes text data, and that\'s th
en used to generate a custom image. We\'ve seen models for code with produc
ts like Copilot that can help complete code as it\'s being authored. And IB
M\'s innovating across all of these domains. So whether it\'s language mode
ls that we\'re building into products like Watson Assistant and Watson Disc
overy, vision models that we\'re building into products like Maximo Visual
Inspection, or Ansible code models that we\'re building with our partners a
t Red Hat under Project Wisdom. We\'re innovating across all of these domai
ns and more. We\'re working on chemistry. So, for example, we just publishe
d and released molformer, which is a foundation model to promote molecule d
iscovery or different targeted therapeutics. And we\'re working on models f
or climate change, building Earth Science Foundation models using geospatia
l data to improve climate research. I hope you found this video both inform
ative and helpful. If you\'re interested in learning more, particularly how
IBM is working to improve some of these disadvantages, making foundation mo
dels more trustworthy and more efficient, please take a look at the links b
elow. Thank you.'
```

Adding the `fetch_transcript` tool to your tools list.

In [13]:
```
tools.append(fetch_transcript)
```

Defining YouTube search tool

Now let's create a search tool that allows finding videos on YouTube based on a query string. This tool uses the `Search` class from the PyTube library to perform searches on YouTube. When given a search term, it returns a list of matching videos with each video represented as a dictionary containing the title, video ID, and a shortened URL. This tool will be helpful for discovering relevant videos when you don't already have a specific URL in mind.

In [14]:
```
from pytube import Search
from langchain.tools import tool
from typing import List, Dict

@tool
def search_youtube(query: str) -> List[Dict[str, str]]:
    """
    Search YouTube for videos matching the query.

    Args:
        query (str): The search term to look for on YouTube

    Returns:
        List of dictionaries containing video titles and IDs in format:
        [{'title': 'Video Title', 'video_id': 'abc123'}, ...]
        Returns error message if search fails
    """
    try:
        s = Search(query)
        return [
            {
                "title": yt.title,
                "video_id": yt.video_id,
                "url": f"https://youtu.be/{yt.video_id}"
            }
            for yt in s.results
        ]
    except Exception as e:
        return f"Error: {str(e)}"
```

Now, you'll test your `search_youtube` tool by calling it with the `.run()` method and the search query "Generative AI." This will return a list of YouTube videos related to generative AI.

In [15]:
```
search_out=search_youtube.run("Generative AI")
display(JSON(search_out))
```

```
<IPython.core.display.JSON object>
```

Appending the `search_youtube` tool to tools list.

In [16]:
```
tools.append(search_youtube)
```

Defining metadata extraction tool

Now you'll create a tool that extracts detailed metadata from a YouTube video using the `yt-dlp` library. This tool takes a YouTube URL and returns comprehensive information about the video, including its title, view count, duration, channel name, like count, comment count, and any chapter markers.

In [17]:
```
@tool
def get_full_metadata(url: str) -> dict:
    """Extract metadata given a YouTube URL, including title, views, duratic
    with yt_dlp.YoutubeDL({'quiet': True, 'logger': yt_dpl_logger}) as ydl:
        info = ydl.extract_info(url, download=False)
        return {
            'title': info.get('title'),
            'views': info.get('view_count'),
            'duration': info.get('duration'),
            'channel': info.get('uploader'),
            'likes': info.get('like_count'),
            'comments': info.get('comment_count'),
            'chapters': info.get('chapters', [])
        }
```

Now, you'll test your `get_full_metadata` tool by running it on a specific YouTube video URL. This will extract comprehensive information about the video with ID "qWHaMrR5WHQ" without downloading the actual video content.

In [ ]:
```
meta_data=get_full_metadata.run("https://youtu.be/qWHaMrR5WHQ")
display(JSON(meta_data))
```

Adding the `get_full_metadata` tool to your tools list.

In [ ]:
```
tools.append(get_full_metadata)
```

Defining trending videos tool

Now, you'll create a tool to fetch the currently trending videos on YouTube for a specific region. This tool uses `yt-dlp` to access YouTube's trending feed based on a provided country code (like "US" for the United States or "IN" for India). It collects important information about each trending video, including the title, video ID, URL, channel name, duration, and view count.

In [ ]:
```
from langchain.tools import tool

from typing import List, Dict

@tool
def get_trending_videos(region_code: str) -> List[Dict]:
    """
    Fetches currently trending YouTube videos for a specific region.
```

```
    Args:
        region_code (str): 2-letter country code (e.g., "US", "IN", "GB")

    Returns:
        List of dictionaries with video details: title, video_id, channel, v
    """
    ydl_opts = {
        'geo_bypass_country': region_code.upper(),
        'extract_flat': True,
        'quiet': True,
        'force_generic_extractor': True,
        'logger': yt_dpl_logger
    }

    try:
        with yt_dlp.YoutubeDL(ydl_opts) as ydl:
            info = ydl.extract_info(
                'https://www.youtube.com/feed/trending',
                download=False
            )

            trending_videos = []
            for entry in info['entries']:
                video_data = {
                    'title': entry.get('title', 'N/A'),
                    'video_id': entry.get('id', 'N/A'),
                    'url': entry.get('url', 'N/A'),
                    'channel': entry.get('uploader', 'N/A'),
                    'duration': entry.get('duration', 0),
                    'view_count': entry.get('view_count', 0)
                }
                trending_videos.append(video_data)

            return trending_videos[:25]  # Return top 25 trending videos

    except Exception as e:
        return [{'error': f"Failed to fetch trending videos: {str(e)}"}]
```

Now, you'll test your `get_trending_videos` tool by running it with the region code `"US"` to fetch trending videos from the United States.

In [ ]:
```
trending_videos=get_trending_videos.run("US")
# Display as formatted JSON
display(JSON(trending_videos))
```

Now, let's add the `get_trending_videos` tool to your tools list.

In [ ]:
```
tools.append(get_trending_videos)
```

Defining thumbnail retrieval tool

Now you'll create a tool to extract all available thumbnail images for a YouTube video. This tool uses `yt-dlp` to retrieve information about the various thumbnail images that YouTube generates for videos at different resolutions. For each thumbnail, collect its URL, width, height, and formatted resolution.

In [ ]:
```
@tool
def get_thumbnails(url: str) -> List[Dict]:
    """
    Get available thumbnails for a YouTube video using its URL.

    Args:
        url (str): YouTube video URL (any format)

    Returns:
        List of dictionaries with thumbnail URLs and resolutions in YouTube'
    """

    try:
        with yt_dlp.YoutubeDL({'quiet': True, 'logger': yt_dpl_logger}) as y
            info = ydl.extract_info(url, download=False)

            thumbnails = []
            for t in info.get('thumbnails', []):
                if 'url' in t:
                    thumbnails.append({
                        "url": t['url'],
                        "width": t.get('width'),
                        "height": t.get('height'),
                        "resolution": f"{t.get('width', '')}x{t.get('height'
                    })

            return thumbnails

    except Exception as e:
        return [{"error": f"Failed to get thumbnails: {str(e)}"}]
```

Now, you'll test your `get_thumbnails` tool by running it on a specific YouTube video URL. This will extract information about all available thumbnail images for the video.

In [ ]:
```
thumbnails=get_thumbnails.run("https://www.youtube.com/watch?v=qWHaMrR5WHQ")

display(JSON(thumbnails))
```

Now, let's add the `get_thumbnails` tool to your tools list.

In [ ]:
```
tools.append(get_thumbnails)
```

Binding tools

Now, you'll bind your collection of tools to the language model. It enables the LLM to access and use your custom YouTube tools during conversations. By binding the tools,

you're giving the model the ability to call these functions when it determines they're needed to fulfill a user request, making the LLM aware of your tools' capabilities and how to use them.

In [ ]:
```
llm_with_tools = llm.bind_tools(tools)
```

The `bind_tools()` function passes all this information to the language model. It converts each tool's attributes (name, description, parameters schema) into a standardized format that the LLM can understand and use to determine when and how to call specific tools based on user requests. Similar to the following code where the schema for each tool is stored:

In [ ]:
```
for tool in tools:
    schema = {
   "name": tool.name,
   "description": tool.description,
   "parameters": tool.args_schema.schema() if tool.args_schema else {},
   "return": tool.return_type if hasattr(tool, "return_type") else None}
    display(JSON(schema))
```

How the LLM calls a tool

Now, define a sample user query that asks for a summary of a specific YouTube video. This query will be used to demonstrate how your LLM can understand a natural language request and use the appropriate tools you've provided to fulfill it.

In [ ]:
```
query = "I want to summarize youtube video: https://www.youtube.com/watch?v=
print(query)
```

Repeating a message object to represent your user query. You'll be wrapping the query string in a HumanMessage object, which is the standard way to format user inputs in LangChain. It represents a human message as a person is expected to initiate the interaction.

In [ ]:
```
messages = [HumanMessage(content = query)]
print(messages)
```

LangChain tool binding process

This step involves sending your message to the LLM and storing its response. Here you'll invoke the language model with your user query about summarizing a YouTube video. The response will contain both text content and potentially tool calls that the model decides to make. `response_1` contains the LLM's response to the user message, including any tool calls it decides to make. The response object contains the content of

the LLM's reply plus structured information about which tools it wants to call and with what parameters.

In [ ]:
```
response_1 = llm_with_tools.invoke(messages)
response_1
```

Adding the LLM's response to your conversation history. After receiving the response from the language model (which contains the tool call to extract the video ID), append it to your messages list to maintain the conversation context. This builds up the chat history that will be used for subsequent interactions with the model.

In [ ]:
```
messages.append(response_1)
```

Extracting tool call information

After receiving the LLM's response, you need to extract the structured tool call information. The line tool_calls_1 = response_1.tool_calls gets the tool call objects that contain which tool the LLM has decided to use and what parameters to pass to it. This information will be used to execute the appropriate tool with the correct inputs.

Creating a tool mapping dictionary

Now you'll create a dictionary that maps tool names to their corresponding function objects. This mapping will be useful later when you need to programmatically invoke specific tools based on their names. It allows you to easily look up and execute a tool function when you have only the tool name as a string, which will be important when processing tool calls from the language model.

In [ ]:
```
tool_mapping = {
    "get_thumbnails" : get_thumbnails,
    "get_trending_videos": get_trending_videos,
    "extract_video_id": extract_video_id,
    "fetch_transcript": fetch_transcript,
    "search_youtube": search_youtube,
    "get_full_metadata": get_full_metadata
}
```

Extracting the tool calls from the language model's response. When the LLM determines it needs to use one of your tools, it includes structured "tool_calls" in its response. Here, you're accessing those tool calls to see which tools the model decided to use in order to fulfill the request about summarizing the YouTube video.

In [ ]:
```
tool_calls_1 = response_1.tool_calls
display(JSON(tool_calls_1))
```

Here you're seeing the structure of the tool call that the LLM decided to make. The tool call is formatted as a dictionary with the following key components:

1. `name` : 'extract_video_id' - This identifies which tool the LLM wants to use first (the video ID extraction tool)
2. `args` : Contains the arguments to pass to the tool - in this case, the YouTube URL from your query
3. `id` : A unique identifier for this specific tool call, which helps track the request/response pair
4. `type` : Indicates this is a tool call rather than other types of AI responses

This shows that the LLM correctly understood it needs to first extract the video ID from the URL before it can proceed with summarizing the video content.

Accessing the name of the first tool that the LLM decided to use. Here you're extracting just the name component `('extract_video_id')` from the first tool call in the list.

In [ ]:
```
tool_name=tool_calls_1[0]['name']
print(tool_name)
```

You need a tool ID to help the LLM know where the output came from:

In [ ]:
```
tool_call_id =tool_calls_1[0]['id']
print(tool_call_id)
```

Accessing the arguments that need to be passed to the chosen tool. Here, you're extracting the arguments component from the first tool call, which contains the YouTube URL that needs to be processed.

In [ ]:
```
args=tool_calls_1[0]['args']
print(args)
```

Adding the LLM's response to your conversation history. After receiving the response from the language model (which contains the tool call to extract the video ID), you append it to your messages list to maintain the conversation context. This builds up the chat history that will be used for subsequent interactions with the model.

Executing the tool call that the LLM requested. Here, you're using your tool mapping dictionary to:

1. Look up the appropriate function based on the tool name ('extract_video_id')
2. Call that function with the arguments provided by the LLM
3. Capture the output (the extracted video ID)

This shows how you can programmatically execute the tools that the LLM decided to use. First, you get the tool from `tool_mapping` .

In [ ]:
```
my_tool=tool_mapping[tool_calls_1[0]['name']]
```

You'll then call the tool with the arguments:

In [ ]:
```
video_id =my_tool.invoke(tool_calls_1[0]['args'])
video_id
```

Adding the tool's output to your conversation history. You'll create a `ToolMessage` that contains:

1. The result from executing the tool (the extracted video ID)
2. The original tool call ID to link this response back to the specific request

By appending this message to your conversation history, you're informing the LLM about the results of the tool execution, which it can use in its next response.

In [ ]:
```
messages.append(ToolMessage(content = video_id, tool_call_id = tool_calls_1[
```

Send your updated conversation to the LLM and store its new response. Now that you've informed the model about the extracted video ID, invoke it again to continue the process. The model will see both the original query and the result of the video ID extraction, allowing it to determine the next step needed to summarize the YouTube video.

In [ ]:
```
response_2 = llm_with_tools.invoke(messages)
response_2
```

The result is a AI messege! Send your updated conversation to the LLM and store its new response. Now that you've informed the model about the extracted video ID, you'll invoke it again to continue the process. The model will see both the original query and the result of the video ID extraction, allowing it to determine the next step needed to summarize the YouTube video.

In [ ]:
```
messages.append(response_2)
```

Extracting the tool calls from the language model's second response. After receiving the video ID, the LLM will likely decide to use another tool to help with the summarization task.

In [ ]:
```
tool_calls_2 = response_2.tool_calls
tool_calls_2
```

Here, you can see that the LLM has decided to use the `fetch_transcript` tool as its next step.

The model is passing two arguments to the transcript fetching tool:

1. `video_id` : 'T-D1OfcDW1M' - The ID that was extracted from the original YouTube URL
2. `language` : 'en' - Requesting the transcript in English as specified in the user's query

---

Fetching the transcript using the video ID obtained in the previous step. Here, you're executing the second tool that the LLM requested by:

1. Looking up the appropriate function `('fetch_transcript')` from your tool mapping
2. Invoking it with the video ID and language parameters
3. Storing the resulting transcript content

In [ ]:
```
fetch_transcript_tool_output = tool_mapping[tool_calls_2[0]['name']].invoke(
fetch_transcript_tool_output
```

You're adding the transcript content to your conversation history by creating another `ToolMessage` that contains the transcript text and the ID of the tool call that requested it. This gives the LLM access to the actual video content so it can generate a summary.

In [ ]:
```
messages.append(ToolMessage(content = fetch_transcript_tool_output, tool_cal
```

Generating the final summary by sending your complete conversation history to the LLM. Now that the model has access to both the video ID and the full transcript, you'll invoke it one more time to generate the summary that the user requested.

In [ ]:
```
summary = llm_with_tools.invoke(messages)
```

In [ ]:
```
summary
```

Automating the tool calling process

You manually saw how you input a text request to your LLM, where the LLM recognized that a tool call was required. Then, you extracted the tool content, formatted the input, made the next tool call, and repeated these steps. While this step-by-step approach helps understand the process, it would be tedious to implement for every application. Now let's automate this entire workflow.

Extracting tool information from LLM response

Create a function to automate tool calling. The input is the tool call object from which you extract the name, and use the tool_mapping dictionary to find the correct function to call. You'll pass the arguments from the tool call to this function and then send the output back as a ToolMessage with the tool_call_id included. The tool_call_id is an essential part of this process as it links each tool response back to the specific tool request made by the language model. This ID ensures the LLM can match responses to

its requests, which is crucial when multiple tools are called in sequence or simultaneously. Without this ID, the LLM would have no way to know which response corresponds to which request, making multi-step reasoning impossible.

In [ ]:
```python
# Define the processing steps
def execute_tool(tool_call):
    """Execute single tool call and return ToolMessage"""
    try:
        result = tool_mapping[tool_call["name"]].invoke(tool_call["args"])
        return ToolMessage(
            content=str(result),
            tool_call_id=tool_call["id"]
        )
    except Exception as e:
        return ToolMessage(
            content=f"Error: {str(e)}",
            tool_call_id=tool_call["id"]
        )
```

You are now going to chain all your functions or tools together, but before you do so, you need to format the data properly. Not only are you required to store the output of each tool, but you also need to store state information like tool IDs. To do this effectively, you must ensure the output of each tool can be properly passed to the next step in your pipeline. The RunnablePassthrough component allows you to maintain state throughout the chain while adding or transforming data at each step, making it ideal for connecting your various tools into a cohesive workflow. The RunnableLambda, placed at the end of your chain, serves a different purpose - it extracts only the final result you want to present to the user. After all the tool calls and message processing, you have a rich state object with many fields, but the user typically only needs the final answer. The RunnableLambda transforms this complete state into just the information you want to return.

In [ ]:
```python
from langchain_core.runnables import RunnablePassthrough, RunnableLambda
```

Building the summarization chain

Now, you'll combine your functions into a complete `summarization_chain` using the pipe operator `|`, which applies functions sequentially (similar to function composition where `f|g(x)` is equivalent to `f(g(x))`).

The workflow follows these steps:

1. Convert the input prompt to a HumanMessage
2. Pass the message to LLM with tools
3. Extract tool calls from LLM response

4. Update message history with tool results
5. Send updated messages back to LLM
6. Repeat steps 3-5 as needed
7. Finally, extract just the content from the final message using RunnableLambda

Each step maintains state using RunnablePassthrough until you reach the final message, at which point you'll apply RunnableLambda to extract only the summary text.

In [ ]:
```
summarization_chain = (
    # Start with initial query
    RunnablePassthrough.assign(
        messages=lambda x: [HumanMessage(content=x["query"])]
    )
    # First LLM call (extract video ID)
    | RunnablePassthrough.assign(
        ai_response=lambda x: llm_with_tools.invoke(x["messages"])
    )
    # Process first tool call
    | RunnablePassthrough.assign(
        tool_messages=lambda x: [
            execute_tool(tc) for tc in x["ai_response"].tool_calls
        ]
    )
    # Update message history
    | RunnablePassthrough.assign(
        messages=lambda x: x["messages"] + [x["ai_response"]] + x["tool_mess
    )
    # Second LLM call (fetch transcript)
    | RunnablePassthrough.assign(
        ai_response2=lambda x: llm_with_tools.invoke(x["messages"])
    )
    # Process second tool call
    | RunnablePassthrough.assign(
        tool_messages2=lambda x: [
            execute_tool(tc) for tc in x["ai_response2"].tool_calls
        ]
    )
    # Final message update
    | RunnablePassthrough.assign(
        messages=lambda x: x["messages"] + [x["ai_response2"]] + x["tool_mes
    )
    # Generate final summary
    | RunnablePassthrough.assign(
        summary=lambda x: llm_with_tools.invoke(x["messages"]).content
    )
    # Return just the summary text
    | RunnableLambda(lambda x: x["summary"])
)
```

Here's how you invoke the summarization chain with a YouTube video URL; this passes your query containing a YouTube URL to the chain, which automatically extracts the video ID, fetches the transcript, and generates a summary of the content.

In [ ]:
```
# Usage
result = summarization_chain.invoke({
    "query": "Summarize this YouTube video: https://www.youtube.com/watch?v=
})

print("Video Summary:\n", result)
```

Up to this point, you've demonstrated how to manually orchestrate the tool calling process step by step. You first invoked the LLM with the user's query, interpreted its decision to use the `extract_video_id` tool, executed that tool, fed the result back to the LLM, processed its next decision to use the `fetch_transcript` tool, executed that tool, and finally had the LLM generate a summary based on the transcript.

Now you'll see how to accomplish the same workflow more efficiently using LangChain's chain functionality, which automates this back-and-forth process of tool selection, execution, and response handling.

Creating the initial message setup

Here you're setting up the first step of your chain that will handle the initial user query. The `RunnablePassthrough.assign` creates a component that takes an input dictionary containing a "query" and converts it into a list containing a single `HumanMessage` object.

In [ ]:
```
initial_setup = RunnablePassthrough.assign(
    messages=lambda x: [HumanMessage(content=x["query"])]
)
```

Defining the first LLM interaction

Here, you'll create the second step of your chain, which handles the first interaction with the language model. This component takes the formatted messages from the previous step, sends them to your tool-equipped LLM, and captures the response in a field called "ai_response."

In [ ]:
```
first_llm_call = RunnablePassthrough.assign(
    ai_response=lambda x: llm_with_tools.invoke(x["messages"])
)
```

Processing the first tool call

Here, you're defining the processing step that handles the LLM's first tool call. This component:

1. Executes each tool call by passing it to your `execute_tool` function, which runs the appropriate tool and returns the result as a `ToolMessage`

2. Updates the message history by combining the original messages, the LLM's response, with the tool calls, and the tool results
3. Prepares the updated conversation state for the next interaction with the LLM

In [ ]:
```
first_tool_processing = RunnablePassthrough.assign(
    tool_messages=lambda x: [
        execute_tool(tc) for tc in x["ai_response"].tool_calls
    ]
).assign(
    messages=lambda x: x["messages"] + [x["ai_response"]] + x["tool_messages
)
```

Defining the second LLM interaction

Here, you're creating the next step in your chain that handles the second interaction with the language model. This component takes the updated message history (which now includes the results from the first tool call) and sends it to the LLM again.

In [ ]:
```
second_llm_call = RunnablePassthrough.assign(
    ai_response2=lambda x: llm_with_tools.invoke(x["messages"])
)
```

Processing the second tool call

Here, you're defining the processing step that handles the LLM's second tool call. Similar to the first tool processing step, this component executes the tool calls (typically fetching the transcript), creates tool messages with the results, and updates the message history by combining everything for the final summarization step.

In [ ]:
```
second_tool_processing = RunnablePassthrough.assign(
    tool_messages2=lambda x: [
        execute_tool(tc) for tc in x["ai_response2"].tool_calls
    ]
).assign(
    messages=lambda x: x["messages"] + [x["ai_response2"]] + x["tool_message
)
```

Generating the final summary

Here, you're defining the final step that produces the summary of the YouTube video. This component:

1. Takes the complete message history (which now contains the original query, tool calls, and tool results)
2. Invokes the LLM one last time to generate a summary
3. Extracts just the content field from the LLM's response
4. Uses a RunnableLambda to return only the summary text as the final output

In [ ]:
```
final_summary = RunnablePassthrough.assign(
    summary=lambda x: llm_with_tools.invoke(x["messages"]).content
) | RunnableLambda(lambda x: x["summary"])
```

Assembling the complete chain

Now, you're combining all the individual components you've defined into a single cohesive chain. By piping each step to the next, you'll create a workflow that:

1. Formats the initial query
2. Gets the first LLM response (video ID extraction)
3. Processes the first tool call
4. Gets the second LLM response (transcript request)
5. Processes the second tool call
6. Generates the final summary

In [ ]:
```
chain = (
    initial_setup
    | first_llm_call
    | first_tool_processing
    | second_llm_call
    | second_tool_processing
    | final_summary
)
```

Now, you're testing your automated chain with the original video summarization query you handled manually before. By passing in the same query to your chain, you can confirm that it produces the same results but in a much more streamlined manner.

In [ ]:
```
query = {"query": "I want to summarize youtube video: https://www.youtube.co
result = summarization_chain.invoke(query)
print("Video Summary:\n", result)
```

Testing the Chain with a Different Query

Here, you're testing your completed chain with a new query to demonstrate its flexibility. Instead of requesting a video summary, you're asking for information about trending videos in India. You'll create a dictionary with the query and invoke your chain, which will handle all the necessary tool calls automatically.

In [ ]:
```
query = {"query": "Get top 3 youtube videos in India and their metadata"}
result = summarization_chain.invoke(query)
print("Video Summary:\n", result)
```

In [ ]:
```
result
```

Recursive chain flow

Now that you've created a chain that works well for your specific two-step tool calling process, you need to consider more complex scenarios. Your current chain is limited to exactly two tool calls in a fixed sequence. In real-world applications, you might need a variable number of tool calls depending on the user's query - for example, getting trending videos and then fetching metadata for each video, or searching for videos on a topic and then getting transcripts for multiple results.

To handle these more complex scenarios, you'll build a recursive chain that can dynamically decide how many tool calls are needed and continue processing until all necessary information has been gathered.

In [ ]:
```python
from langchain_core.runnables import RunnableBranch, RunnableLambda
from langchain_core.messages import HumanMessage, ToolMessage
import json

def execute_tool(tool_call):
    """Execute single tool call and return ToolMessage"""
    try:
        result = tool_mapping[tool_call["name"]].invoke(tool_call["args"])
        content = json.dumps(result) if isinstance(result, (dict, list)) els
    except Exception as e:
        content = f"Error: {str(e)}"

    return ToolMessage(
        content=content,
        tool_call_id=tool_call["id"]
    )
```

Defining the core processing logic

This function handles the core processing logic of your recursive chain. It takes the current conversation history and:

1. Identifies the most recent message in the conversation
2. Extracts all tool calls from that message and executes them in parallel using your `execute_tool` helper
3. Updates the message history by adding the tool response messages
4. Gets the next response from the language model based on the updated conversation
5. Returns the complete updated message history with both tool responses and the new LLM response

In [ ]:
```python
def process_tool_calls(messages):
    """Recursive tool call processor"""
    last_message = messages[-1]

    # Execute all tool calls in parallel
    tool_messages = [
        execute_tool(tc)
```

```
        for tc in getattr(last_message, 'tool_calls', [])
    ]

    # Add tool responses to message history
    updated_messages = messages + tool_messages

    # Get next LLM response
    next_ai_response = llm_with_tools.invoke(updated_messages)

    return updated_messages + [next_ai_response]
```

Creating the recursive stopping condition

This function determines whether your recursive process should continue or terminate. It:

1. Takes the current message history and examines the last message
2. Checks if that message contains any tool calls using the `getattr` function (which safely handles cases where the attribute might not exist)
3. Returns a boolean value - `True` if there are more tool calls to process, and `False` when you reach a point where the LLM has provided a final answer without requesting additional tools

In [ ]:
```
def should_continue(messages):
    """Check if you need another iteration"""
    last_message = messages[-1]
    return bool(getattr(last_message, 'tool_calls', None))
```

Implementing the recursive function

This function implements the actual recursion that powers your dynamic tool calling process:

1. It first checks the stopping condition using the `should_continue` function to determine if more tools need to be called
2. If more tool calls are needed, it processes those calls using your `process_tool_calls` function and then recursively calls itself with the updated messages
3. If no more tool calls are needed, it returns the final message history, which contains the complete conversation, including the LLM's final response

After defining this recursive function, you'll wrap it in a `RunnableLambda` to make it compatible with LangChain's chain architecture.

In [ ]:
```
def _recursive_chain(messages):
    """Recursively process tool calls until completion"""
    if should_continue(messages):
        new_messages = process_tool_calls(messages)
        return _recursive_chain(new_messages)
```

```
    return messages

recursive_chain = RunnableLambda(_recursive_chain)
```

Building the complete universal chain

Now, you're assembling your final universal chain that can handle any type of query requiring any number of tool calls. This chain consists of three main steps:

1. The first step converts the user query into a properly formatted `HumanMessage` object
2. The second step sends this initial message to your tool-equipped LLM and adds the LLM's first response to the message history
3. The final step passes the conversation to your recursive chain, which will handle all subsequent tool calls until the LLM provides a final answer

This universal chain is much more flexible than your earlier fixed-step chain, as it can dynamically adapt to queries that require different numbers and types of tool calls.

In [ ]:
```
universal_chain = (
    RunnableLambda(lambda x: [HumanMessage(content=x["query"])])
    | RunnableLambda(lambda messages: messages + [llm_with_tools.invoke(mess
    | recursive_chain
)
```

In [ ]:
```
print(universal_chain.invoke({
    "query": "Show top 3 US trending videos with metadata and thumbnails"
})[-1])
```

Exercise

Exercise 1: Try a different video with a Youtube link

In [ ]:
```
# TODO
```

▶ Click here for hint

Exercise 2: Extract the video ID

In [ ]:
```
# TODO
```

▶ Click here for hint

Exercise 3: Collect all necessary data about the video in one go

In [ ]:
```
# TODO
```

▶ Click here for hint

Exercise 4: Get video transcript

In [ ]:
```
# TODO
```

▶ Click here for hint

Exercise 5: Get video thumbnails

In [ ]:
```
# TODO
```

▶ Click here for hint

Let's have a comprehensive prompt to be passed to LLM to generate a summary

In [ ]:
```
# TODO
```

▶ Click here for prompt

Exercise 6: Single LLM invocation with all the data

In [ ]:
```
# TODO
```

▶ Click here for hint

Exercise 7: Display the comprehensive analysis

In [ ]:
```
# TODO
```

▶ Click here for hint

Authors

Kunal Makwana is a Data Scientist at IBM and is currently pursuing his Master's in Computer Science at Dalhousie University.

Other Contributors

Joseph Santarcangelo